

.NET MAUI **Projects**

A guide to building applications for Windows,
macOS, Android, and iOS



Sivaraj Ambat



.NET MAUI **Projects**

A guide to building applications for Windows,
macOS, Android, and iOS



Sivaraj Ambat



.NET MAUI Projects

*A guide to building
applications for
Windows, macOS, Android,
and iOS*

Sivaraj Ambat



www.bpponline.com

First Edition 2025

Copyright © BPB Publications, India

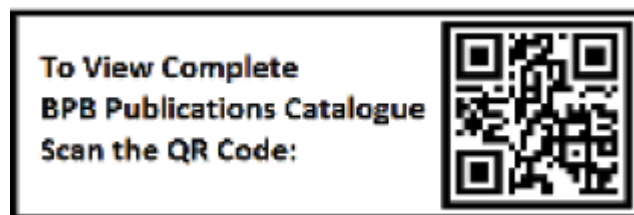
ISBN: 978-93-65899-924

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

My family

About the Author

Sivaraj Ambat is a seasoned software professional with expertise in web, mobile and desktop app development, cloud technologies, and artificial intelligence. With over 30 years of experience across multiple industries, he has worked in sectors such as instrumentation, industrial automation, software development, and education.

Sivaraj began his career working on industrial embedded systems and robotics, focusing on automation solutions in the textile industry before transitioning into software development and training. Over the years, he has built extensive experience in developing software across a range of platforms, including web, mobile and desktop. He possesses deep expertise in a variety of programming languages, frameworks, and tools, and consistently keeps up with the latest industry trends and technological advancements.

A recipient of the **2016 Top 20 Emerging Leaders** award from *Training Magazine* and **Senior Fellowship Award** from his previous employer, Sivaraj has made significant contributions to the field of software development. He has also authored and reviewed several books and articles and frequently delivers talks on a wide range of topics related to software development.

About the Reviewer

Aleksei Starkov is a software engineer specializing in .NET, Xamarin, and .NET MAUI technologies, with extensive experience in building advanced cross-platform mobile applications and frameworks. Over the years, Aleksei has worked with a diverse range of companies, from Agile startups to large-scale enterprises, contributing to innovative, high-quality projects with millions of users that leverage modern technologies.

Aleksei is also the author of *Learning .NET MAUI: Unlock the Potential of .NET MAUI for Cross-Platform App Development*, a comprehensive resource designed to help developers understand .NET MAUI and learn how to develop modern cross-platform production-oriented applications. As a technical reviewer, Aleksei brings his extensive real-world expertise and keen attention to detail, offering valuable insights to help ensure technical content is accurate, practical, and useful for its audience.

With his deep knowledge of cross-platform development and his passion for empowering developers, Aleksei continues to make a meaningful impact in the field of mobile application development. His contributions help both individuals and teams to unlock the full potential of .NET MAUI.

Acknowledgement

I want to express my heartfelt gratitude to everyone who played a role in the completion of this book.

First and foremost, I am deeply thankful to my family, friends, and colleagues at DXC Technology for their unwavering support and encouragement throughout this journey. Their encouragement and belief in me have been a constant source of motivation.

I also want to express my deep gratitude to my managers for their unwavering encouragement and support. Vidya Gothe, Director - Global Talent Transformation at DXC Technology, has been an incredible source of motivation, always encouraging and supporting me to develop my technical expertise further and inspiring me to pursue writing. I would also like to thank Prabhakar Kanagarajan, Director - Applications at DXC Technology, whose mentorship and encouragement played a valuable role in shaping my skills and providing invaluable guidance throughout my professional journey. Their belief in my abilities has been a constant source of strength and has contributed greatly to the successful completion of this book.

I am profoundly grateful to BPB Publications for their expertise and support in bringing this book to life. Their guidance was invaluable in navigating the complexities of the publishing process.

My sincere appreciation goes to the reviewers, technical experts, and editors who provided valuable feedback and helped refine the manuscript. Their insights and suggestions have significantly enhanced the quality of the book.

Lastly, I want to thank the readers who have shown interest in this book. Your support and enthusiasm mean a great deal to me.

Preface

In today's rapidly advancing technological landscape, the ability to develop cross-platform applications is becoming increasingly important for developers. **.NET MAUI Projects** is designed to introduce you to the world of cross-platform app development using .NET MAUI, a powerful and flexible framework for building mobile applications at no cost.

This book is structured to guide you step-by-step, assuming no prior knowledge of .NET MAUI or mobile app development. The first five chapters lay the foundation, beginning with an introduction to cross-platform app development and .NET MAUI, followed by setting up your development environment. We then dive into the core programming languages used in .NET MAUI development - C# and XAML - ensuring that you are equipped with the necessary skills to develop apps.

The remaining five chapters focus on hands-on projects, providing detailed, step-by-step guidance on how to build real-world applications. By the end of these projects, you will not only have developed several apps but you will also be well-equipped to create your own cross-platform applications and take your skills to the next level.

This book is written for absolute beginners, whether you are a student, a working professional seeking to learn this skill, or simply an enthusiast eager to learn. Through clear explanations, practical examples, and accessible language, I aim to make .NET MAUI app development approachable and engaging.

I hope that this book will serve as a valuable tool in your journey to becoming proficient in building cross-platform mobile applications. With the skills gained from these chapters, you will be well on your way to creating versatile apps for Windows, MacOS, Android and iOS, opening up new opportunities in the world of app development.

Chapter 1: Introduction to Cross-platform Application Development

- In this chapter, we will discuss the basics of cross-platform application development. We will start by understanding the need for cross-platform

app development and the problems that it helps address. We will discuss the advantages and limitations of cross-platform app development and why it has emerged as a preferred choice for many developers and organizations, examining its advantages, such as code reusability, faster time-to-market, and broader audience reach. Then, we will explore various cross-platform development frameworks and tools available for developers. We will also discuss the impact of emerging technologies on cross-platform app development and its future.

Whether you are a seasoned developer seeking to broaden your skill set or a newcomer eager to explore the possibilities of cross-platform development, this chapter offers a comprehensive roadmap to navigate the exciting and ever-evolving landscape of multi-platform application development.

Chapter 2: Overview of .NET MAUI - In this chapter, we will begin with a high-level overview of .NET MAUI, exploring its underlying architecture and the key principles that drive its functionality. We will gain an understanding of how .NET MAUI works under the hood (meaning we are going to be looking at the internal workings of .NET MAUI to understand how it functions at an architectural level) and how it empowers developers to leverage its capabilities effectively.

Additionally, we will cover a whirlwind tour of the common **application programming interfaces (APIs)** provided by .NET MAUI, which help developers create dynamic and responsive cross-platform UIs. This chapter will provide a foundation for the readers to understand the flexibility and versatility inherent in .NET MAUI, which will help while developing apps in the later chapters of this book.

Furthermore, we will look into the evolution of .NET MAUI from its predecessor, Xamarin. Understanding this transition is crucial for developers looking to migrate their existing Xamarin apps to .NET MAUI seamlessly. We will look into the migration process and the benefits of transitioning to .NET MAUI.

Chapter 3: Development Environment Setup - In this chapter, we will also guide you through configuring your development environment to ensure optimal results. Since .NET MAUI is primarily focused on cross-platform development, our main focus will be on setting up your development environment on the Windows operating system. We will

also discuss the additional steps required for configuring your Android development environment, which is crucial for testing and deployment.

Chapter 4: A Crash Course in C# - This chapter gives a concise and focused introduction to C# designed to equip beginners with the essential knowledge needed to follow the chapters ahead in this book. While this chapter does not provide an exhaustive coverage of the C# language, it serves as a vital primer to ensure that readers, especially those new to C# programming, can comfortably follow and comprehend subsequent discussions. For those seeking in-depth exploration, this publisher offers comprehensive books dedicated to the topic in detail.

Chapter 5: Introduction to XAML - In a .NET MAUI app, the **user interface (UI)** is basically constructed as a tree of objects. The UI is generally created using a markup language instead of C#. This markup language is an XML-based language and is called **eXtensible Application Markup Language (XAML)**. The markup defines the .NET MAUI classes and property values. At runtime, the XAML is parsed and the objects are instantiated and initialized.

Chapter 6: Project-1: Color Picker - In this chapter, we will develop a Color Picker using the concepts we have learned so far. A Color Picker, also called RGB Color Mixer, is an essential tool for graphic designers, artists, and developers who need to select colors for their application. It is found in various graphic design software, image editing applications, and web development tools. The Color Picker allows users to adjust the values of red, green, and blue color components individually or in combination to create desired colors.

Chapter 7: Project-2: Tip Calculator - In this chapter, you will be introduced to creating a Tip Calculator. A Tip Calculator is a tool or app that helps you figure out how much tip to leave after having food at a restaurant. The tip amount varies depending on the country and cultural norms. In many countries, it is customary to leave a tip of around 15% to 20% of the total bill at restaurants. However, some places may have different tipping customs, and some people may choose to leave more or less than the usual percentage based on the quality of service or personal preferences. Also, when dining in a group, it is common to split the total bill, including the tip, equally among all the individuals to ensure a fair distribution of expenses. This way, everyone contributes their share, making it easier to handle the bill and the tip without any confusion or inconvenience. In this chapter, you will learn how to build a Tip

Calculator that will allow users to enter the bill amount, select the desired tip percentage, and the number of people who will split the bill. It will help you instantly calculate the tip amount and the split amount for each user from the total bill amount.

Chapter 8: Project-3: BMI Calculator - A **Body Mass Index (BMI)** calculator is a tool or app used to assess an individual's body weight in relation to their height. It serves as a basic indicator of whether a person's weight falls within a healthy range or deviates from it. BMI is widely used in healthcare, fitness, and wellness programs to provide a simple yet informative snapshot of one's health. The app that we will build in this chapter is not an exhaustive BMI calculator catering to all age groups but rather a simplified version aimed at showcasing key concepts of .NET MAUI. It demonstrates the fundamental functionalities of a BMI calculator, offering users an interactive way to calculate their BMI and receive a basic understanding of their weight status.

Chapter 9: Project-4: Unit Converter - A Unit Converter is an invaluable tool that simplifies the conversion of measurements from one unit to another. Doing these conversions manually can become complex, even using a calculator. Unit converters are especially useful for students and a wide range of professionals in various fields who must perform unit conversions as a part of their day-to-day work. In this chapter, we will develop a Unit Converter app that will help users convert values swiftly and accurately among diverse units of measurement related to length, weight, volume, area, temperature, and time. The focus will be on understanding how to use .NET MAUI to develop such an app and not on applying best practices using design patterns. The application structure is intentionally kept simple so that a beginner can also follow it easily. Advanced readers might want to implement it using the **Model-View-ViewModel (MVVM)** pattern, which provides the benefit of separating the concerns, making the code more maintainable and promotes, and promoting testability. That are beyond our control.

Chapter 10: Project-5: Weather App - Weather is one such aspect that affects our lives in many ways. It decides what we do, wear, carry, and how we travel. A weather app is, therefore, a trusted companion in helping us navigate different weather conditions and ensure that our lives are not affected. Besides knowing about the weather and the forecast, the weather app provides an excellent learning opportunity, as it involves

some of the advanced skills interfacing with REST API, parsing data, and presenting it in a user-friendly format.

In this chapter, we will develop a simple weather app that will allow the user to enter the name of a place and find out the weather, particularly the current temperature and type of weather, and know the weather forecast for the next 7 days, including max and min. temperature. In the process, we will learn how to work with REST API and parse the data. We will also learn how to use the MVVM architectural pattern in application development.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the *Coloured Images* of the book:

<https://rebrand.ly/46df18>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/.NET-MAUI-Projects>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to Cross-platform Application Development

Introduction

Structure

Objectives

Introducing cross-platform app development

Apps versus. websites

App development approaches

Pros and cons of cross-platform app development

Popular development frameworks and technologies

React Native

Flutter

NativeScript

Uno Platform

Avalonia

.NET MAUI

Emerging technologies and cross-platform app development

Conclusion

Points to remember

2. Overview of .NET MAUI

Introduction

Structure

Objectives

Introduction to .NET MAUI

Supported platforms

Understanding how NET MAUI API works

.NET MAUI app anatomy

Pages, Layouts and Views

Pages

Layouts

Views

Evolution of NET MAUI from Xamarin.Forms

Migration from Xamarin.Forms

Manual migration

Assisted migration using .NET upgrade assistant

Platform integration

Conclusion

Points to remember

3. Development Environment Setup

Introduction

Structure

Objectives

Installation of Visual Studio IDE

Testing the environment setup

Debugging on Windows

Debugging on Android Emulator

Debugging on Android devices

iOS device provisioning

Pair to Mac for iOS development

Conclusion

Points to remember

4. A Crash Course in C#

Introduction

Structure

Objectives

Introduction to C#

- Types, variables, and constants
- Strings
- Operators and expressions
- Branches and loops
 - Loops*
 - For Loop*
 - Foreach*
 - While and Do-while*
 - Break and continue*
- Classes, methods, and properties
 - Classes*
 - Methods*
 - Properties*
- Inheritance and interface
 - Interfaces*
- Delegates and events
 - Events*
- Arrays and lists
- Conclusion
- Points to remember

5. Introduction to XAML

- Introduction
- Structure
- Objectives
- Overview of XAML
- XAML syntax
 - Object element syntax*
 - Comments*
 - Attribute syntax*
 - Empty element syntax*
 - Property value syntax*

Attached property syntax

Namespaces

Object names

Markup extensions

Passing arguments

Data binding

Conclusion

Points to remember

6. Project-1: Color Picker

Introduction

Structure

Objectives

Creating the layout

Creating the data entry section

Specifying colors in XAML

Changing the background color

Generating random colors

Copying the color to the clipboard

Conclusion

Points to remember

7. Project-2: Tip Calculator

Introduction

Structure

Objectives

Creating the layout

Creating the data entry section

Applying graphical design principles in UI design

Performing the calculation

Conclusion

Points to remember

8. Project-3: BMI Calculator

Introduction

Structure

Objectives

Creating the layout

Creating a data entry section

Data binding between UI controls

Data binding to an object

Binding value converters

Conclusion

Points to remember

9. Project-4: Unit Converter

Introduction

Structure

Objectives

Creating the main menu layout

Creating the navigational system

Creating the data entry section

Performing the conversions

Conclusion

Points to remember

10. Project-5: Weather App

Introduction

Structure

Objectives

Introduction to MVVM architecture

Creating the project structure and the UI

Designing the user interface

Updating the UI with date
Updating the UI with place name
Using the weather API
Creating the model
Calling the weather API from code
Creating custom type converters
Updating the UI with weather data
Displaying seven-day weather forecast
Updating the weather forecast
Conclusion
Points to remember

Index

CHAPTER 1

Introduction to Cross-platform Application Development

Introduction

In today's digital world, people use all kinds of devices like smartphones, tablets, and computers. They want apps that work well on any device they use. For someone who wants to develop apps that work seamlessly across these different devices, it can be overwhelming to decide which tools and technologies to use. This has made cross-platform app development very important in software engineering.

In this chapter, we will discuss the basics of Cross-platform application development. We will start by understanding the need for cross-platform app development and the problems that it helps address. We will discuss the advantages and limitations of cross-platform app development and why cross-platform application development has emerged as a preferred choice for many developers and organizations, examining its advantages, such as code reusability, faster time-to-market, and broader audience reach. Then we will explore various cross-platform development frameworks and tools that are available for developers. We will also

discuss the impact of emerging technologies on cross-platform app development and its future.

Whether you are a seasoned developer seeking to broaden your skill set or a newcomer eager to explore the possibilities of cross-platform development, this chapter offers a comprehensive roadmap to navigate the exciting and ever-evolving landscape of multi-platform application development.

Structure

In this chapter, we will discuss the following topics:

- Introducing cross-platform app development
- Pros and cons of cross-platform app development
- Popular development frameworks and technologies
- Emerging technologies and cross-platform app development

Objectives

After reading this chapter, you will be able to:

- Understand the different approaches for app development and their respective advantages and disadvantages.
- Appreciate the need for cross-platform app development.
- Learn the advantages and limitations of cross-platform app development.
- Familiarize yourself with some of the popular development frameworks and technologies used for cross-platform app development.
- Acquire insight into the impact of emerging technologies on cross-platform app development and its future.

Introducing cross-platform app development

In today's dynamic digital landscape, software developers are presented with unprecedented opportunities to create applications that cater to the

diverse array of devices used by consumers worldwide. From smartphones and tablets to wearables, smart TVs, and IoT devices, the proliferation of technology has led to a proliferation of platforms, each offering unique capabilities and user experiences.

Apps versus. websites

Consumers and businesses alike are increasingly drawn to apps over websites for their unparalleled convenience, personalized experiences, and seamless integration with the digital ecosystem. Apps offer users instant access to services and information, eliminating the need to navigate through browser windows or bookmarks. With features like push notifications, offline access, and device-specific functionalities, apps provide a more immersive and engaging experience, fostering deeper connections with users. Moreover, apps can leverage device capabilities such as GPS, cameras, and sensors to offer enhanced functionality and context-aware services. For businesses, apps offer a direct channel to engage with customers, build brand loyalty, and drive revenue through in-app purchases and subscriptions. With the rise of mobile-first lifestyles and the ubiquity of smartphones and tablets, apps have become indispensable tools for both consumers and businesses, providing a seamless and intuitive way to interact in the digital age.

The ever-evolving digital landscape and consumer expectations provide developers with infinite opportunities to innovate and develop solutions that enhance productivity, entertainment, communication, and more across a vast range of devices. As the boundaries between physical and digital worlds blur, software developers find themselves at the forefront of innovation, leveraging their creativity and technical prowess to build applications that seamlessly integrate into the fabric of everyday life, enriching experiences and shaping the future of technology.

App development approaches

Over the years, the proliferation of technology has led to an explosion of development frameworks and tools for app development, which can be overwhelming for developers as they navigate the choices available to them to develop apps. To simplify this landscape and offer clarity and guidance, we can categorize app development frameworks into five primary types. Let us look at them in detail:

- **Native apps:** Native applications are developed for a specific platform, utilizing the native programming languages and **application programming interface (APIs)** provided by the platform's ecosystem. While native apps offer unparalleled performance and access to platform-specific features, they require separate development efforts for each platform, resulting in increased time and resources. For example, Android applications are created using Kotlin or Java in the Android Studio **integrated development environment (IDE)** and require the Android SDK for development, and apps are distributed through the Play Store. iOS applications are created using Swift in Xcode IDE and distributed through the App Store. Windows applications are created using C# and Visual Studio IDE and distributed through the Windows Marketplace.
- **Web apps:** Web applications leverage web technologies such as HTML, CSS, and JavaScript to deliver content through web browsers across various platforms. While web apps offer broad accessibility and easy deployment, they often lack the performance and capabilities of native apps, particularly for complex functionalities and interactions. Websites developed using **responsive web design (RWD)** principles with a mobile-first approach are often categorized under the Web Apps category, providing accessible and optimized experiences across different devices. **progressive web applications (PWAs)** are also a popular approach for app development within the web apps category, offering native-like experiences and enhanced usability across various devices.
- **Hybrid apps:** Hybrid applications combine elements of native and web development, utilizing web technologies within a native shell. This approach enables developers to leverage existing web development skills while accessing native device features. However, compared to fully native solutions, hybrid apps may suffer from performance issues and limited access to platform-specific features. Popular hybrid app frameworks are Electron, Apache Cordova/PhoneGap, Ionic and Blazor, which blend native and web development technologies in their codebase to deliver cross-platform functionality.

- **Cross-platform apps:** Cross-platform applications are a newer category where apps for all platforms are developed using a single source code and programming language. When compiled, they mostly generate native code for each respective platform. This provides a compelling solution, offering the promise of code reusability and streamlined development processes across multiple platforms. By using frameworks and tools designed specifically for cross-platform development, developers can write code once and deploy it across various platforms, reducing development time and costs significantly.
- **Low code/no code apps:** The **low code/no code (LCNC)** approach to app development represents a radical shift in the software development landscape, offering a solution for creating applications without extensive coding expertise. With this approach, individuals with varying levels of technical proficiency can design, develop, and deploy applications using intuitive visual interfaces, drag-and-drop components, and pre-built templates. By abstracting away much of the traditional coding complexities, LCNC platforms empower users to focus on solving business problems and achieving their objectives more efficiently. The limitations of the LCNC approach include potential constraints on customization, scalability, and complexity handling, which may not always meet the requirements of highly specialized or intricate application projects. Popular LCNC platforms include Microsoft PowerApps, Google AppSheet, Salesforce Lightning Platform, Mendix, Glide, etc.

Pros and cons of cross-platform app development

Cross-platform app development offers the following advantages:

- **Code reusability:** Cross-platform development allows developers to write code once and deploy it across multiple platforms, saving time and resources by avoiding the need to develop separate codebases for each platform.
- **Faster time-to-market:** With cross-platform development, developers can simultaneously target multiple platforms, reducing the time it takes to launch their app and reach a broader audience.

- **Cost-effectiveness:** By leveraging cross-platform development frameworks and tools, developers can significantly reduce development costs compared to building separate native apps for each platform.
- **Broad audience reach:** Cross-platform apps can be deployed across various platforms, including iOS, Android, and web browsers, maximizing the potential audience and market reach for the app.

However, cross-platform app development also has a few limitations, though these are general and not necessarily applicable to all frameworks. Let us look at them in detail:

- **Lack of support from native platform creators:** Some of the cross-platform app development frameworks are not directly supported by native platform creators. Sometimes, this can cause cross-platform app development to lag behind, as not all frameworks quickly catch up to support the latest native API versions. This can result in delays in accessing new features and functionalities available on specific platforms.
- **Performance limitations:** Cross-platform apps may suffer from performance issues compared to native apps, as they often rely on abstraction layers and additional overhead to achieve platform compatibility.
- **Limited access to platform-specific features:** Cross-platform development frameworks may not provide access to all platform-specific features and functionalities, restricting the app's capabilities compared to fully native solutions.
- **Increased complexity:** Managing codebases across multiple platforms and ensuring consistent performance and user experience can add complexity to cross-platform development projects, requiring careful planning and implementation.
- **Dependency on third-party tools:** Cross-platform development often relies on third-party frameworks and tools, introducing dependencies and potential compatibility issues that developers must manage throughout the app's lifecycle.

Overall, while cross-platform development offers significant advantages regarding code reusability, time-to-market, and cost-effectiveness, developers should carefully weigh these benefits against potential performance limitations and platform-specific constraints to determine the most suitable approach for their app development projects.

Popular development frameworks and technologies

In the dynamic landscape of cross-platform app development, choosing the right framework or platform is crucial for achieving success. This section explores some of the most popular frameworks and platforms that empower developers to build robust and versatile applications that run seamlessly across multiple devices and operating systems. However, it is important to note that the landscape of cross-platform app development is rapidly evolving, and the information presented here may change over time. Let us look at the popular frameworks and platforms in detail in the upcoming sub-sections.

React Native

React Native is an open-source framework developed by *Facebook* for building cross-platform mobile applications using JavaScript and React. It originated from *Facebook's* internal hackathon project, aiming to address the challenges of building mobile apps efficiently across multiple platforms. React Native follows the principles of React, a popular JavaScript library for building user interfaces, by enabling developers to create components that render native UI elements.

To work with React Native, developers need proficiency in JavaScript and React and an understanding of mobile app development concepts. Familiarity with native development concepts and APIs is beneficial but not required.

React Native offers the following advantages:

- **Code reusability:** React Native allows developers to write code once and deploy it across multiple platforms, reducing development time and effort.
- **Native performance:** React Native bridges the gap between JavaScript and native code, enabling apps to achieve near-native performance.

- **Hot reloading:** React Native offers a hot reloading feature, allowing developers to instantly see changes in the app during development without rebuilding the entire project.
- **Rich ecosystem:** React Native has a vibrant ecosystem with a wide range of third-party libraries and tools, enabling developers to extend the functionality of their apps easily.

React Native also has certain limitations too. Let us look at them in detail:

- **Platform-specific features:** React Native may not support all platform-specific features out-of-the-box, requiring developers to write custom native modules or use third-party libraries.
- **Debugging challenges:** Debugging React Native apps can be challenging, especially when dealing with issues related to bridging between JavaScript and native code.
- **Performance variability:** While React Native apps can achieve near-native performance, there may be performance variations across different devices and platforms.
- **Rapid changes:** React Native is evolving rapidly, with frequent updates and changes to APIs, which may require developers to adapt and update their codebase continuously.

Overall, React Native offers a powerful and efficient solution for building cross-platform mobile applications, leveraging the flexibility of JavaScript and the performance of native development. However, developers should be aware of its limitations and be prepared to address them to ensure a successful app development experience.

Flutter

Flutter is an open-source UI software development kit created by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. It originated from Google's internal project called **Sky**, which aimed to develop a portable UI framework. Flutter follows the principles of reactive programming, enabling developers to create visually rich, fast, and intuitive user interfaces.

To work with Flutter, developers need proficiency in the Dart programming language, as Flutter apps are primarily written in Dart. Familiarity with object-oriented programming concepts and UI design principles is required. Additionally, knowledge of platform-specific APIs and development tools may be necessary for certain functionalities.

Flutter offers the following advantages:

- **Single codebase:** Flutter allows developers to write code once and deploy it across multiple platforms, including iOS, Android, web, and desktop, saving time and effort.
- **Fast development:** Flutter offers a hot reload feature, allowing developers to see changes in real-time without restarting the app, speeding up the development process.
- **Expressive UI:** Flutter provides a rich set of customizable widgets and animations, enabling developers to create visually appealing and interactive user interfaces. Widgets are the building blocks of the UI, which define physical aspects like text and buttons, and layout aspects such as padding and alignment.
- **Native performance:** Flutter compiles code to native machine code, resulting in high-performance apps with smooth animations and fast startup times.

Flutter also has a few limitations. Let us look at them in detail:

- **Limited third-party libraries:** Flutter's ecosystem is still evolving, and it may lack certain third-party libraries and tools available in more mature frameworks.
- **Large app size:** Flutter apps may have larger file sizes compared to native apps, as they include the Flutter engine and framework, which can impact download and installation times.
- **Platform-specific features:** While Flutter provides access to many platform-specific features, some advanced functionalities may require additional native code integration.
- **Learning curve:** Flutter's reactive programming model and Dart language may have a learning curve for developers unfamiliar with these concepts, especially those coming from other programming backgrounds.

Overall, Flutter offers a powerful and versatile solution for building cross-platform applications with expressive UIs and native performance. While Flutter has some limitations, it continues to gain popularity among developers for its fast development cycles and impressive UI capabilities.

NativeScript

NativeScript is an open-source framework developed by Progress for building cross-platform mobile applications using JavaScript or TypeScript and Angular. It originated at Telerik, which was later acquired by Progress. NativeScript follows the principles of native development, enabling developers to access platform-specific APIs and UI components directly from JavaScript or TypeScript code.

To work with NativeScript, developers need proficiency in JavaScript or TypeScript and Angular, depending on their preferred development approach. Familiarity with native mobile app development concepts and APIs is beneficial, as NativeScript allows direct access to platform-specific features.

NativeScript offers the following advantages:

- **Native performance:** NativeScript enables developers to build truly native mobile apps using JavaScript or TypeScript, resulting in high-performance applications with access to platform-specific features and functionalities.
- **Access to Native APIs:** NativeScript provides direct access to platform-specific APIs and UI components, allowing developers to leverage the full capabilities of each platform.
- **Code reusability:** With NativeScript, developers can reuse code across multiple platforms, reducing development time and effort.
- **Rich ecosystem:** NativeScript has a vibrant ecosystem with a wide range of plugins, libraries, and tools, enabling developers to extend the functionality of their apps easily.

There are some known limitations, too:

- **Learning curve:** NativeScript's learning curve may be steep for developers new to mobile app development, especially those unfamiliar with JavaScript or TypeScript.

- **Limited UI components:** While NativeScript provides access to platform-specific UI components, its selection may be limited compared to other frameworks, requiring developers to create custom UI elements for certain functionalities.
- **Plugin compatibility:** Some third-party plugins may not be compatible with NativeScript or require additional configuration to work properly, which can add complexity to the development process.
- **Performance optimization:** Achieving optimal performance in NativeScript apps may require careful optimization and testing, especially for complex applications with high processing and rendering requirements.

Overall, NativeScript offers a powerful and flexible solution for building cross-platform mobile applications with native performance and access to platform-specific features. While it has its limitations, NativeScript continues to be a popular choice among developers for its ability to deliver high-quality mobile experiences across multiple platforms.

Uno Platform

Uno Platform is an open-source framework for building cross-platform applications using C# and XAML, targeting iOS, Android, Windows, WebAssembly, macOS, and Linux. It originated from the UWP Community Toolkit project and was later developed into a full-fledged cross-platform framework by the Uno Platform team. Uno Platform follows the principles of the **Universal Windows Platform (UWP)**, enabling developers to create visually rich and interactive user interfaces using XAML and leverage the power of the .NET ecosystem.

To work with the Uno Platform, developers need proficiency in C# and XAML, as well as familiarity with the .NET ecosystem. Experience with UWP development concepts and APIs is beneficial but not required. Additionally, knowledge of platform-specific features and development tools may be necessary for certain functionalities.

Uno Platform provides the following advantages:

- **Single codebase:** Uno Platform allows developers to write code once and deploy it across multiple platforms, including iOS,

Android, Windows, WebAssembly, macOS, and Linux, maximizing code reuse and reducing development time and effort.

- **Native performance:** Uno Platform apps compile to native code, resulting in high-performance applications with smooth animations and fast startup times on each platform.
- **Access to platform-specific APIs:** Uno Platform provides access to platform-specific APIs and capabilities, enabling developers to create apps that integrate seamlessly with each platform.
- **Rich UI components:** Uno Platform offers a rich set of UI controls and components, allowing developers to create visually appealing and responsive user interfaces using XAML.

There are some known limitations, too:

- **Learning curve:** Uno Platform's learning curve may be steep for developers new to C# and XAML, especially those coming from other programming backgrounds.
- **Platform-specific customization:** While Uno Platform enables cross-platform development, developers may need to write platform-specific code or use platform-specific features to achieve certain functionalities or optimizations.
- **Ecosystem maturity:** Uno Platform's ecosystem is still evolving, and some features or functionalities may be less mature or require additional development effort compared to other frameworks.
- **Tooling support:** While Uno Platform integrates with Visual Studio and other development tools, some platform-specific features or development workflows may require additional setup or configuration.

Overall, Uno Platform offers a powerful and flexible solution for building cross-platform applications with native performance and access to platform-specific features. While it has its limitations, Uno Platform continues to gain popularity among developers for its ability to deliver high-quality cross-platform experiences using familiar C# and XAML development paradigms.

Avalonia

Avalonia is an open-source, cross-platform UI framework for .NET, designed to build native user interfaces using XAML and C#. It originated from the .NET Foundation and is maintained by a community of contributors. Avalonia follows the principles of UWP, enabling developers to create visually rich and interactive user interfaces that run on multiple platforms, including Windows, macOS, Linux, Android and iOS.

To work with Avalonia, developers need proficiency in C# and XAML, as well as familiarity with the .NET ecosystem. Experience with desktop application development concepts and **Model-View-ViewModel (MVVM)** architecture is beneficial but not required. Additionally, knowledge of platform-specific features and development tools may be necessary for certain functionalities.

Avalonia provides the following advantages:

- **Cross-platform compatibility:** Avalonia allows developers to build native desktop applications that run on Windows, macOS, and Linux platforms, maximizing code reuse and reducing development effort.
- **XAML-based UI development:** Avalonia uses **eXtensible Application Markup Language (XAML)** to define user interfaces, providing a familiar and expressive markup language for designing UI layouts and components.
- **Native performance:** Avalonia compiles code to native machine code, resulting in high-performance applications with smooth animations and fast startup times on each platform.
- **Extensible ecosystem:** Avalonia has a growing ecosystem of third-party libraries, controls, and tools, enabling developers to extend the functionality of their applications easily.

There are a few limitations with Avalonia framework, too. Let us look at some of them:

- **Learning curve:** Avalonia's learning curve may be steep for developers new to desktop application development or XAML-based frameworks, especially those coming from other programming backgrounds.

- **Limited third-party support:** While Avalonia has a growing ecosystem, it may lack certain third-party libraries and tools available in more mature frameworks, requiring developers to implement certain functionalities themselves.
- **Platform-specific differences:** While Avalonia aims to provide a consistent development experience across platforms, there may be platform-specific differences or limitations that developers need to consider when building cross-platform applications.

The main difference between Avalonia UI and .NET MAUI is in its approach to rendering user interfaces. Avalonia UI utilizes a custom drawing engine based on Skia, whereas .NET MAUI relies on the native UI components of each platform.

Overall, Avalonia offers a powerful and flexible solution for building cross-platform desktop applications with native performance and access to platform-specific features. While it has its limitations, Avalonia continues to gain popularity among developers for its ability to deliver high-quality desktop experiences using familiar C# and XAML development paradigms.

.NET MAUI

.NET Multi-platform App UI (MAUI) is an open-source framework developed by Microsoft for building cross-platform applications using .NET and C#, targeting iOS, Android, macOS, and Windows platforms. It originated from Xamarin.Forms and is designed to provide a unified and modern approach to cross-platform app development. .NET MAUI follows the principles of the Xamarin ecosystem, enabling developers to create native user interfaces with a single codebase and a familiar development experience.

To work with .NET MAUI, developers need proficiency in C# and .NET, as well as familiarity with XAML for defining user interfaces. Experience with Xamarin.Forms or other XAML-based frameworks are beneficial but not required. Additionally, knowledge of platform-specific features and APIs may be necessary for certain functionalities.

.NET MAUI is chosen as the subject for this book because it represents the future of cross-platform app development with .NET. It builds upon the success of Xamarin.Forms and introduces new capabilities and

improvements for building modern, native applications across multiple platforms. Developers must consider .NET MAUI because it offers a unified and powerful framework for building cross-platform applications with native performance, a familiar development experience, and seamless integration with the .NET ecosystem. As the successor to Xamarin.Forms, .NET MAUI provides a strategic advantage for developers looking to leverage their existing skills and investments in .NET to build cross-platform apps that reach a broader audience.

Emerging technologies and cross-platform app development

Emerging technologies are revolutionizing cross-platform app development, paving the way for more efficient, feature-rich, and innovative applications. Generational AI and tools like Copilot are empowering developers with advanced capabilities, making them more productive and enabling them to learn and adapt quickly to evolving requirements. By leveraging AI-driven assistance, developers can streamline their workflows, generate code more efficiently, and overcome complex challenges with greater ease. However, the ultimate leader in this rapidly evolving landscape will be the technology that not only delivers optimal performance and supports native app features but also aids developers in implementing quickly, shortening time-to-market, and providing a supportive ecosystem.

With Microsoft's substantial investment in .NET MAUI, we can be confident in the longevity and success of businesses aiming to achieve their goals through cross-platform app development, leveraging the robust capabilities and extensive ecosystem of the .NET platform.

Conclusion

In conclusion, this chapter has provided a comprehensive overview of application development methodologies, emphasizing the importance of cross-platform development in today's digital landscape. We have also explored the benefits and constraints of cross-platform application development. We discussed the popular development frameworks for cross-platform application development and their respective skills requirements, advantages, and disadvantages. As technology continues to evolve and with emerging trends like AI-assisted coding, it is easier for

developers to leverage cross-platform app development to bring their ideas to market quickly while minimizing costs, as opposed to developing in silos using the respective platform's native app development stack. Armed with this understanding, you should be well-prepared to adapt and innovate using .NET MAUI for cross-platform app development!

In the next chapter, we will cover a high-level overview of .NET MAUI, including its architecture and core principles to help you utilize its features effectively. We will also discuss the common APIs for creating dynamic, cross-platform **user interface (UIs)** setting the stage for more advanced app development in subsequent chapters.

Points to remember

Here are some key points from this chapter:

- Native app development is inherently time-consuming, expensive, and lacks code reusability, as each platform requires its own development stack.
- Web and hybrid approaches offer broad compatibility and easier development and maintenance with web development skills but may sacrifice performance and, especially in the case of web apps, limited access to native features compared to native apps.
- Cross-platform application development stands out by offering versatility to develop native applications that seamlessly run across multiple platforms using the same codebase.
- Cross-platform application development bridges the gap between native, web, hybrid and low-code/no-code approaches while maximizing efficiency and reducing development time and costs.
- .NET MAUI is preferred for cross-platform app development due to its robust features, unified development experience, and seamless integration with the .NET ecosystem, streamlining the development of high-performance applications across various platforms.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[**https://discord.bpbonline.com**](https://discord.bpbonline.com)



CHAPTER 2

Overview of .NET

MAUI

Introduction

Welcome to the exciting world of **.NET Multi-platform App UI (.NET MAUI)**. As technology advances and many types of devices proliferate, the need for cross-platform development solutions becomes increasingly important. .NET MAUI emerges as a comparatively young but powerful and stable platform in this landscape, offering developers a streamlined approach to building stunning **user interfaces (UI)** that run seamlessly across multiple platforms.

In this chapter, we will begin with a high-level overview of .NET MAUI, exploring its underlying architecture and the key principles that drive its functionality. We will gain an understanding of how .NET MAUI works under the hood (meaning we are going to be looking at the internal workings of .NET MAUI to understand how it functions at an architectural level) and how it empowers developers to leverage its capabilities effectively.

Additionally, we will cover a whirlwind tour of the common **application programming interfaces (APIs)** provided by .NET MAUI, which help developers create dynamic and responsive cross-platform UIs. This chapter will provide a foundation for the readers to understand the

flexibility and versatility inherent in .NET MAUI, which will help while developing apps in the later chapters of this book.

Furthermore, we will look into the evolution of .NET MAUI from its predecessor, Xamarin. Understanding this transition is crucial for developers looking to migrate their existing Xamarin apps to .NET MAUI seamlessly. We will look into the migration process and the benefits of transitioning to .NET MAUI.

Structure

In this chapter, we will discuss the following topics:

- Introduction to .NET MAUI
- Supported platforms
- Understanding how NET MAUI API works
- .NET MAUI app anatomy
- Evolution of NET MAUI from Xamarin.Forms
- Migration from Xamarin.Forms
- Platform integration

Objectives

After reading this chapter, you will be able to:

- Understand the foundational concepts of .NET MAUI.
- Understand how .NET MAUI works under the hood.
- Exhibit familiarity with the most commonly used APIs in .NET MAUI.
- Learn how .NET MAUI evolved from its predecessor Xamarin.
- Understand platform-specific integration concepts.

Introduction to .NET MAUI

.NET MAUI is an open-source, cross-platform framework designed for developing native mobile and desktop applications using C# and XAML.

Unlike traditional development approaches where separate codebases are required for each platform, .NET MAUI enables developers to create apps that seamlessly run on Android, iOS, macOS, and Windows from a single shared codebase.

Under the hood, .NET MAUI unifies the APIs of Android, iOS, macOS, and Windows into a unified API. Thus, developers have a write-once-run-anywhere experience. At the same time, it also provides developers with deep access to each native platform's functionalities, providing maximum flexibility and control.

Along with .NET Framework 6.0, Microsoft introduced a set of platform-specific frameworks, namely .NET Android, .NET iOS, .NET macOS, and **Windows UI 3 (WinUI 3)** library. These frameworks share a common .NET **Base Class Library (BCL)**, which abstracts platform-specific details from the code. Mono serves as the .NET runtime environment for apps running in Android, iOS, and macOS. For Windows, .NET CoreCLR powers the execution environment.

Although the BCL facilitates sharing common business logic across different platforms, each platform has its own unique way of defining user interfaces and interaction models. Traditionally, developers had to maintain separate codebases for each platform-specific family of devices. .NET MAUI eliminates the need to maintain multiple codebases by providing a unified framework for building UIs for mobile and desktop apps. In a .NET MAUI app, developers primarily interact with the .NET MAUI API, which directly communicates with native platform APIs. Occasionally, developers may need to implement platform-specific features. In such cases, they can utilize methods within the platform-specific framework, as indicated in the following figure:

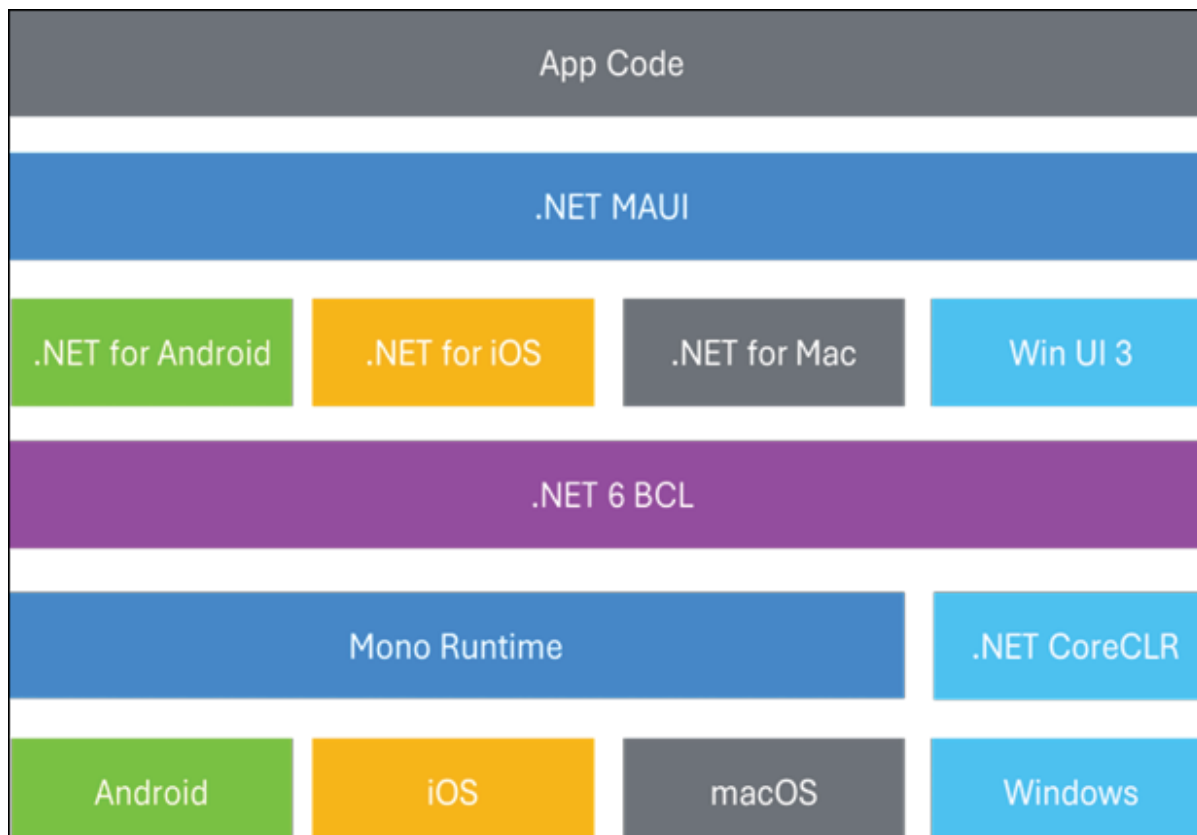


Figure 2.1: .NET MAUI architecture

.NET MAUI apps can be developed on both PC and Mac platforms and compiled into native app packages. Let us understand this in detail:

- Android apps are compiled from C# into an **intermediate language (IL)** and then **just-in-time (JIT)** compiled into native assembly when launched.
- iOS apps are fully **ahead-of-time (AOT)** compiled from C# into native ARM (a semi-conductor technology company) assembly code.
- macOS apps utilize Mac Catalyst, which converts iOS apps built with UIKit into desktop apps augmented with additional AppKit and platform APIs.
- Windows apps leverage the Windows UI 3 (WinUI 3) library to create native desktop apps targeting the Windows platform.

Using .NET MAUI, developers can streamline app development, reduce overhead, and deliver seamless user experiences across a variety of devices and platforms.

Supported platforms

.NET MAUI apps can be developed for the following platforms:

- Android 5.0 (API 21) or higher
- iOS 11 or higher
- macOS 10.15 or higher
- Windows 11 and Windows 10 version 1809 or higher
- Tizen, an **operating system (OS)** developed by Samsung, is also supported

.NET MAUI Blazor apps, however, have the following platform requirements:

- Android 7.0 (API 24) or higher
- iOS 14 or higher
- macOS 11 or higher

Understanding how NET MAUI API works

.NET MAUI separates the implementation of UI elements from their logical descriptions. The UI is described using XAML, which is an XML-based, platform-neutral language. This way we can achieve the abstraction of the UI from the logical implementation. For example, let us consider the following XAML snippet used to create a button control:

1. `<Button Text="Click me"`
2. `Clicked="OnCounterClicked"`
3. `HorizontalOptions="Center" />`

In the preceding code snippet, we have defined the button's label to display the string (**Click me**). It is also specified that on clicking the button, it will trigger the method **OnCounterClicked**. We can also assign additional properties for the button control to modify its layout and text, with text formatting options like emphasizing the text or centering the text for better visual accessibility.

Alternatively, you have the option to dynamically create the UI using C# code. This method allows you to create dynamic layouts based on the

context. For example, you could choose not to display certain controls if the user does not have the necessary authorization level.

Under the hood, .NET MAUI optimizes performance by generating native code tailored to the target device. This is achieved through platform-specific handlers for each UI element and platform. For example, targeting iOS will map .NET MAUI code to an iOS UIButton, while on Android, it will correspond to an Android AppCompatButton. These handlers are indirectly accessed via control-specific interfaces provided by .NET MAUI, such as IButton for buttons. Refer to the following figure for an illustration of how the handlers map the native views to the UI elements:

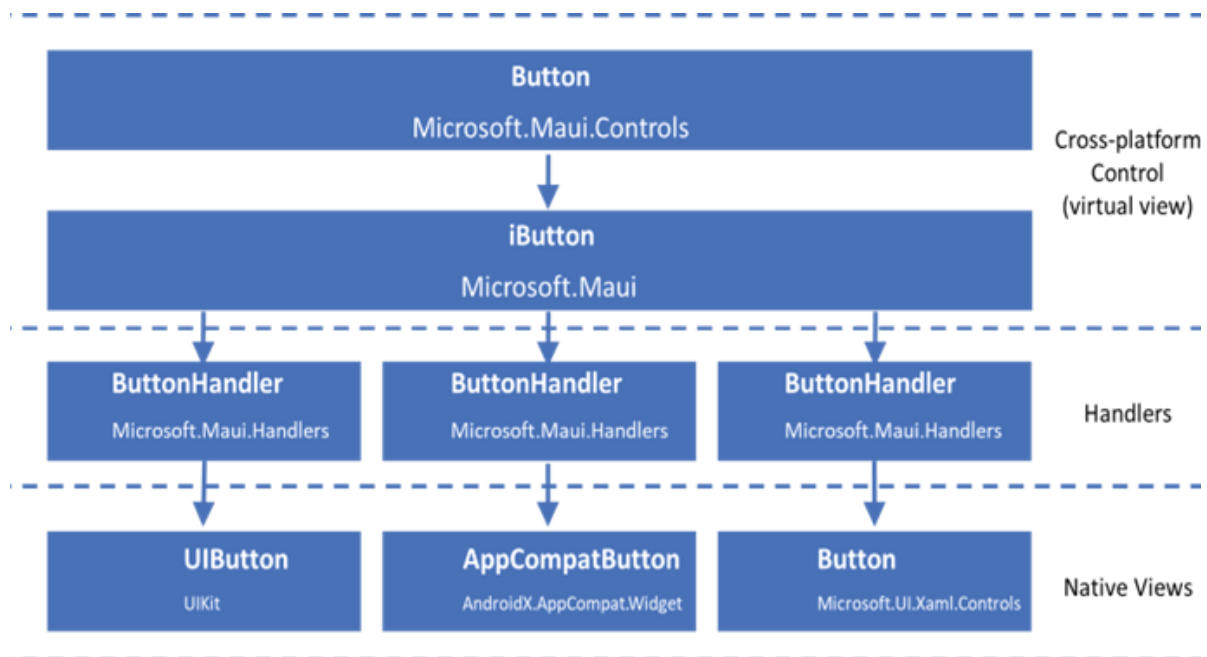


Figure 2.2: Mapping of UI elements to native views

.NET MAUI simplifies access to common controls like buttons, as well as other essential elements such as text entry fields, labels, and date pickers. We will discuss the commonly used controls next. Beyond individual controls, .NET MAUI also enriches app development by providing the following:

- A sophisticated layout engine that will help to create intricate designs of pages.
- Various page types will facilitate diverse navigation patterns, including navigation drawers, which is a common feature in most mobile apps today.

- Support for data binding.
- Custom handler creation to refine the presentation of UI elements.
- Direct access to native APIs and abstraction of common mobile and desktop app needs apart from the UI. The essentials library empowers apps to interact with device features like GPS, accelerometer, battery, and network states, along with numerous other sensors and services essential for mobile development.
- Promoting the use of elegant and maintainable development practices.

.NET MAUI app anatomy

The user interface of a .NET MAUI App is constructed using objects, also known as controls, that map to the native controls of each target platform. The controls can be classified into three different categories:

- Pages
- Layouts
- Views

Refer to the following figure for an illustration of a .NET MAUI app structure:

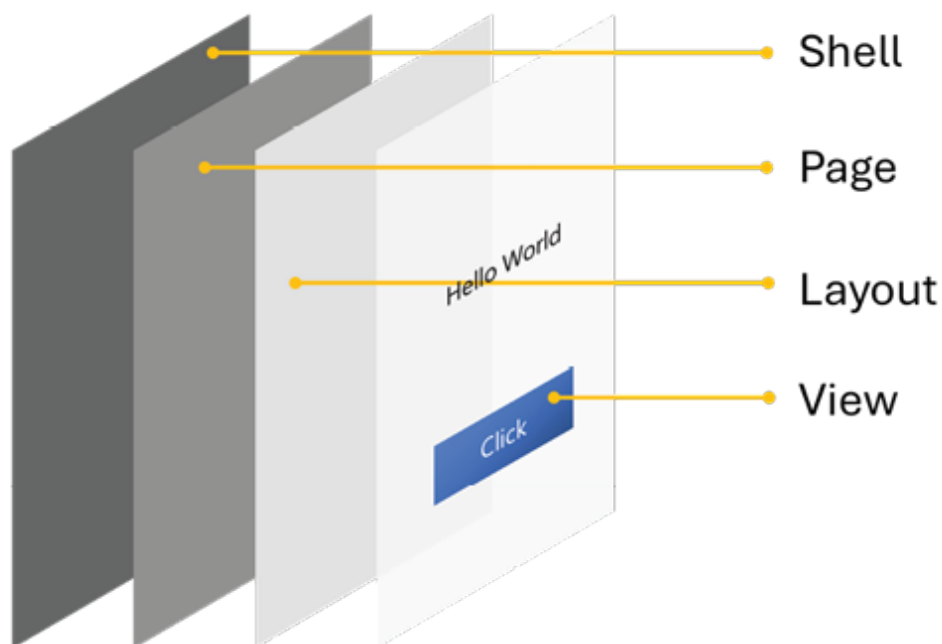


Figure 2.3: .NET MAUI app structure

Pages, Layouts and Views

In this section, we will explore the foundational components of .NET MAUI applications Pages, Layouts, and Views. Each plays a crucial role in structuring the user interface, enabling developers to create intuitive and responsive designs.

Pages

All .NET MAUI applications contain one or more pages. Generally, each screen or window in a .NET MAUI app consists of one page, which occupies the entire screen or window. The page can have only one content, and in order to design a screen with multiple controls, we normally use a Layout as the child of the Page.

.NET MAUI has the following types of Pages:

- **ContentPage**: Displays a single view and is a commonly used page type.
- **FlyoutPage**: Manages two related pages of information—a FlyoutPage presenting items and a detail page providing details about items on the FlyoutPage.
- **NavigationPage**: Offers a hierarchical navigation experience, allowing navigation through pages forwards and backward.
- **TabbedPage**: Contains a series of pages navigable by tabs located at the top or bottom of the page, with each tab loading the corresponding page content.

Layouts

.NET MAUI layouts are used to organize user interface controls into visual arrangements. Each layout typically contains multiple elements, which can be other layouts or views. These layout classes have inherent behaviors for positioning and sizing child elements. .NET MAUI contains the following layouts:

- **Grid**: Arranges child elements in a grid of rows and columns.
- **HorizontalStackLayout**: Arranges child elements in a horizontal stack.
- **StackLayout**: Arranges child elements in either a vertical or horizontal stack.

- **VerticalStackLayout:** Arranges child elements in a vertical stack.
- **BindableLayout:** Allows any layout class to populate its content by binding to a collection of items, with the ability to customize the appearance of each item.
- **FlexLayout:** Enables stacking or wrapping of children with various alignment and orientation options.
- **AbsoluteLayout:** Places child elements at specific locations relative to its parent.

Views

In .NET MAUI, views refer to UI elements like labels, buttons, and sliders, which are commonly referred to as controls or widgets in other environments. .NET MAUI includes different types of views, and for the sake of easy understanding, let us explore the views based on their usage:

Present data or information: These are the options available to present data or information:

- **BoxView:** Draws a rectangle or square with specified height, width and color.
- **Ellipse:** Displays an ellipse or circle.
- **Label:** Displays single-line or multi-line text.
- **Line:** Draws a line between two points.
- **Rectangle:** Draws a rectangle or square.
- **Map:** Displays a map using Microsoft.MauiControls.Maps Nuget package.
- **Polygon:** Draws a polygon.
- **Polyline:** Draws a series of connected lines.
- **Path:** Displays curves and complex shapes.
- **Border:** Container control with borders and backgrounds that can wrap other controls.
- **Image:** Displays an image from local file, URI, or an embedded resource.

- **Frame**: Wraps a view or layout with configurable border, color, shadow, etc.
- **WebView**: Displays web pages or local HTML content.
- **ScrollView**: Provides scrolling for content.

To initiate commands: The following options can be used to initiate commands:

- **Button**: Responds to tap/click for code execution.
- **RadioButton**: Allows selection of one option from a set.
- **SearchBar**: User input control for initiating a search.
- **SwipeView**: Container control with context menu revealed by a swiping gesture.
- **RefreshView**: Provides pull-to-refresh for scrollable content.

To set values: The following options can be used to set values:

- **CheckBox**: Enables selection of Boolean value with checkable button.
- **Slider**: Selects double value from continuous range.
- **Stepper**: Selects double value from incremental range.
- **Switch**: Selects Boolean value with on/off button.
- **DatePicker**: Selects a date with the respective platform's date picker control.
- **TimePicker**: Selects time with the respective platform's time picker control.

To edit text: The following options can be used to edit text:

- **Entry**: Allows entering one or more lines of text.
- **Editor**: Allows entering and editing multiple lines of text.

To indicate activity:

- **ActivityIndicator**: Uses animation to indicate app activity without progress indication.

- **ProgressBar**: Uses animation to show progress through a lengthy app activity.

To display collections: The following options can be used to display collections:

- **CarouselView**: Displays a scrollable list of data items, swiping to navigate.
- **ListView**: Displays a scrollable list of selectable data items.
- **CollectionView**: Displays a scrollable list of selectable data items with different layout specs.
- **TableView**: Displays a table of scrollable items, which can be grouped into sections.

Evolution of NET MAUI from Xamarin.Forms

.NET MAUI is the evolution of Xamarin with better performance, increased productivity, and a more streamlined developer experience. Microsoft declared the end of support for Xamarin with effect from May 1, 2024. After this date, Microsoft will no longer offer bug fixes, security updates, or new features for Xamarin. If you continue to use Xamarin beyond this date, your apps may not be compatible with the latest versions of Android and iOS. Also, deployment to the Android and iOS app stores will no longer be possible.

If you are presently utilizing Xamarin, it is advisable to migrate to .NET MAUI. Microsoft has furnished a roadmap and resources to assist developers in navigating this transition. If you are familiar with Xamarin, transitioning to .NET MAUI will feel intuitive due to its similarities.

It would be good to know how .NET MAUI evolved from Xamarin. For this, you will have to understand a little about the .NET Framework. If someone told you or if you have read anywhere that .NET Framework is proprietary and only meant for applications that can run on Windows OS, it is no longer true. .NET Framework was introduced around 2002 as a proprietary, closed-source framework for developers to be able to develop Windows and Web Applications which can run on Windows OS only. The framework consisted of a large class library called the BCL and a runtime environment called **common language runtime (CLR)**.

The BCL has thousands of ready-to-use classes that provide the functionality to develop the user interface, data access, database connectivity, cryptography, file operations, etc. Some of these classes are versatile, but some of them are specifically meant for certain types of applications and hence grouped into sets and referred to as Windows Forms, WPF, ASP.NET, ADO.NET, Entity Framework, WCF, Workflow Foundation, etc. Some variants of the .NET Framework, like Windows Mobile and Windows CE, were meant for older Windows Mobile and hand-held devices. Developers could use any compatible programming language like VB.NET, or C# or C++ to develop applications and use the classes in the BCL to develop the applications as required. The CLR took care of security, memory management, and exception handling when .NET applications were executed. To develop the applications, Microsoft also provided an IDE called Visual Studio.

The .NET Framework quickly became very popular among developers to develop apps for the Windows environment. To help run .NET applications on other Operating Systems, an open-source project called Mono was initiated by a company called *Ximian*, which was later acquired by Novell in 2003.

In 2011, Novell was acquired by Attachmate, and the developers of Mono moved to form a new company called Xamarin, continued the work, and extended the .NET developer platform with tools and libraries for building apps for Android, iOS, macOS, and Windows **Universal Windows Platform (UWP)**. Using Xamarin, developers could write applications using C# and share the code-base across platforms.

In 2016, Microsoft acquired Xamarin which was bundled with the Visual Studio IDE for developers to create cross-platform applications.

.NET MAUI is an evolution of Xamarin and extends its capabilities from mobile to desktop scenarios. The two architectures share some similarities, but notable distinctions exist between them. Let us look at them in detail:

- Xamarin supports building and running apps with the .NET Framework, while .NET MAUI utilizes the .NET CLI toolchain for building, running, and publishing .NET apps.
- Xamarin employs separate projects for each platform, whereas .NET MAUI uses a unified project structure where each platform can be targeted using devices destined for deployment.

- In Xamarin, platform-dependent files and code are managed within distinct projects, whereas in .NET MAUI, they are organized under platform folders and targeted by platform filename conventions.
- Xamarin requires workarounds for accessing native resources, whereas .NET MAUI offers direct access to native platform APIs.
- Xamarin relies on renderers for creating controls, necessitating the use of custom renderers for customization of native controls, which can impact performance and app size significantly. .NET MAUI adopts a handler architecture that is loosely coupled with native assembly, resulting in a lightweight app with enhanced performance. While renderers are still available in .NET MAUI, they are optional.

In the next section, we will discuss more about migrating to .NET MAUI from Xamarin.

Migration from Xamarin.Forms

Migration from Xamarin.Forms to MAUI can be accomplished through two methods:

- Manual migration
- Assisted migration

Microsoft has published detailed documentation on migrating from Xamarin.Forms to .NET MAUI and it is available on the Microsoft website: <https://learn.microsoft.com/en-us/dotnet/maui/migration/?view=net-maui-8.0>. We will just discuss the high-level process in this section.

Manual migration

The manual migration process involves migrating each file of the Xamarin project to MAUI by creating a new MAUI project. While this method demands significant effort, it is essential for complex and large projects. Before initiating the migration, it is crucial to thoroughly understand the project.

The high-level steps for manual migration are mentioned in the following list:

- Update your Xamarin.Forms app to utilize Xamarin.Forms 5.
- Upgrade the app's dependencies to their latest versions.
- Verify that the app functions correctly.
- Create a .NET MAUI app.
- Transfer code and configuration from the Xamarin.Forms app to the .NET MAUI app.
- Migrate resources from your Xamarin.Forms app to the .NET MAUI app.
- Update namespaces as necessary.
- Address any API changes encountered during the migration process.
- Upgrade or substitute incompatible dependencies with .NET 8 versions.
- Compile and thoroughly test your app to ensure proper functionality.

To streamline the upgrade, it is recommended that you create a new .NET MAUI library project with the same name as the Xamarin.Forms library project and then copy the code, configuration, and resources into it.

Assisted migration using .NET upgrade assistant

The .NET upgrade assistant assists in upgrading Xamarin.Forms projects to .NET MAUI by converting project files and performing common code updates. The Upgrade Assistant performs the following tasks:

- Converts Xamarin.Forms class library, Xamarin.iOS, and Xamarin.Android projects to SDK-style projects.
- Updates the target framework to net8.0-android and net8.0-ios as required.
- Sets `<UseMaui>true</UseMaui>` in project files.
- Adjusts project properties as required.
- Adds/removes specific NuGet packages.

- Removes reference to Xamarin namespaces and replaces them with .NET MAUI namespaces.

The upgrade assistant offers two types of upgrades:

- **In-place:** Upgrades the project without creating a copy.
- **Side-by-side:** Creates a copy of the project and upgrades it, leaving the original project untouched.

Platform integration

In .NET MAUI, each supported platform offers a unique operating system and platform APIs accessible from C#. It provides cross-platform APIs for accessing a wide range of platform functionality, such as sensors, device information, network connectivity, and authentication flows. These APIs are categorized into different areas of functionality. Let us look at them in detail:

- **Application model:** This includes functionalities like managing app actions, accessing app information, launching browsers or maps, and handling permissions and version tracking.
- **Communication:** It covers functionalities related to contacts, email, networking, phone dialer, SMS, and web authentication.
- **Device features:** This section includes functionalities such as battery information, device display metrics, device information, device sensors, flashlight control, geocoding, geolocation, haptic feedback, and vibration.
- **Media:** It provides functionalities like media picking, taking screenshots, text-to-speech, and unit conversion.
- **Sharing:** This involves functionalities related to clipboard operations and sharing files or text.
- **Storage:** It covers functionalities related to file picking, file system operations, preferences storage, and secure storage.

Additionally, .NET MAUI platform-specifics allow for the consumption of specific functionality that is available only on a particular platform. When .NET MAUI lacks specific APIs for accessing platform-specific functionalities, developers can write their own code to access the required platform APIs.

Conclusion

In summary, this chapter served as a comprehensive introduction to .NET MAUI, providing you with the foundational knowledge needed to embark on your cross-platform development journey. Whether you are a seasoned developer or just starting, .NET MAUI offers an exciting opportunity to build robust and innovative applications that can reach a diverse audience across various devices and platforms. Let us unlock the full potential of .NET MAUI together in the subsequent chapters.

In the next chapter, we will discuss setting up your development environment and configuring it for .NET MAUI app development.

Points to remember

- .NET MAUI unifies the APIs of Android, iOS, macOS, and Windows into a unified API.
- Developers can also use platform-specific frameworks, namely .NET Android, .NET iOS, .NET macOS, and Windows UI 3 if required.
- Under the hood, .NET MAUI optimizes performance by generating native code tailored to the target device through platform-specific handlers for each UI element and platform.
- The user interface of a .NET MAUI App is constructed using objects, also known as controls which can be further classified into Pages, Layouts, and Views.
- Cross-platform APIs for accessing platform functionality such as sensors, device information, network connectivity, and authentication flows is supported.
- .NET MAUI is the evolution of Xamarin with better performance, increased productivity, and a more streamlined developer experience.
- Xamarin is no longer supported with effect from 1.5.2024 and migration/upgradation of Xamarin apps is recommended.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Development Environment Setup

Introduction

In this chapter, we will guide you through configuring your development environment to ensure optimal results. Since .NET MAUI is primarily focused on cross-platform development, our main focus will be on setting up your development environment on the Windows operating system. We will also discuss the additional steps required for configuring your Android development environment, which is crucial for testing and deployment.

While we will include details for iOS development, please note that we will not cover it in detail. The setup process for iOS development involves additional steps and requires a macOS system, which is beyond the scope of this chapter. However, we will provide enough information to help you get started with iOS development if you choose to do so in the future. Let us get your development environment up and running smoothly.

Structure

In this chapter, we will discuss the following topics:

- Installation of Visual Studio IDE
- Testing the environment setup
- Debugging on Windows

- Debugging on Android Emulator
- Debugging on Android devices
- iOS device provisioning
- Pair to Mac for iOS development

Objectives

After reading this chapter, you will be able to:

- Setup your development environment
- Debug on Windows environment
- Test the app in Android Virtual and a physical device
- Test the app in iOS simulator or a device
- Learn to pair a Mac to your development environment in Windows for iOS development

Installation of Visual Studio IDE

Visual Studio is a comprehensive IDE, that provides all the tools for everything from writing and editing code to debugging, building, and deploying applications. Alongside these core functionalities, Visual Studio packs compilers, code completion tools, source control features, extensions, and many other features to cover every phase of software development. With Visual Studio, you can write high-quality code efficiently and collaboratively.

Before you begin the installation process, check the prerequisites provided in the documentation. For Visual Studio 2022, the detailed requirements are provided at <https://learn.microsoft.com/en-us/visualstudio/releases/2022/system-requirements>.

Visual Studio is available in three editions tailored to different user needs: Community, Professional, and Enterprise. The Community Edition offers a fully featured IDE for students, open-source projects, and individual developers at no cost, and we will use it for the projects covered in this book. If you are developing for commercial purposes, please ensure that you check the license terms and conditions and opt for the Professional or Enterprise editions as applicable.

In this chapter, the steps are demonstrated for installing Visual Studio 2022 Community Edition on a Windows 10 PC, which has an x64 processor, 8 GB

of RAM, and 50 GB of Hard disk space. You will need administrator permissions on the machine as well.

As Visual Studio is a versatile tool for different kinds of applications, it provides you with a workload-based installer, which allows users to install only the components they need. Follow these steps:

1. You can download the latest version of the Visual Studio Community Edition installer from <https://visualstudio.microsoft.com/vs/community/>. Refer to the following figure for the webpage from where you can download Visual Studio Community Edition:

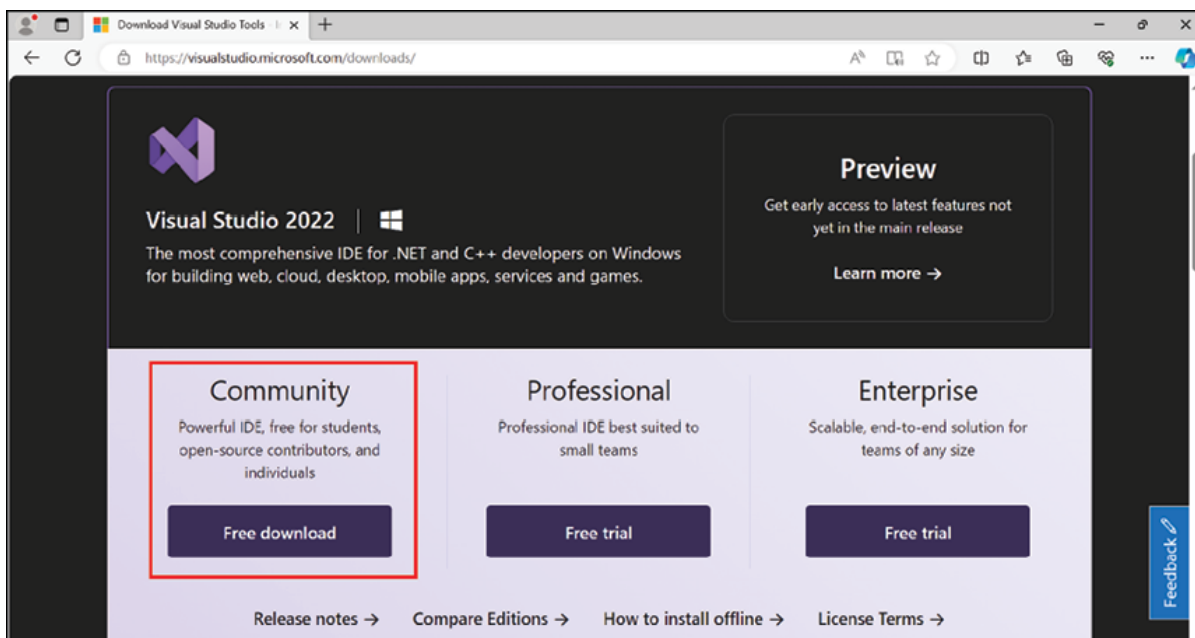


Figure 3.1: Download Visual Studio Community Edition

Unless you have changed the downloads folder path, when the download is complete, the setup file for installation will be saved to your **Downloads** folder, as shown in the following figure:

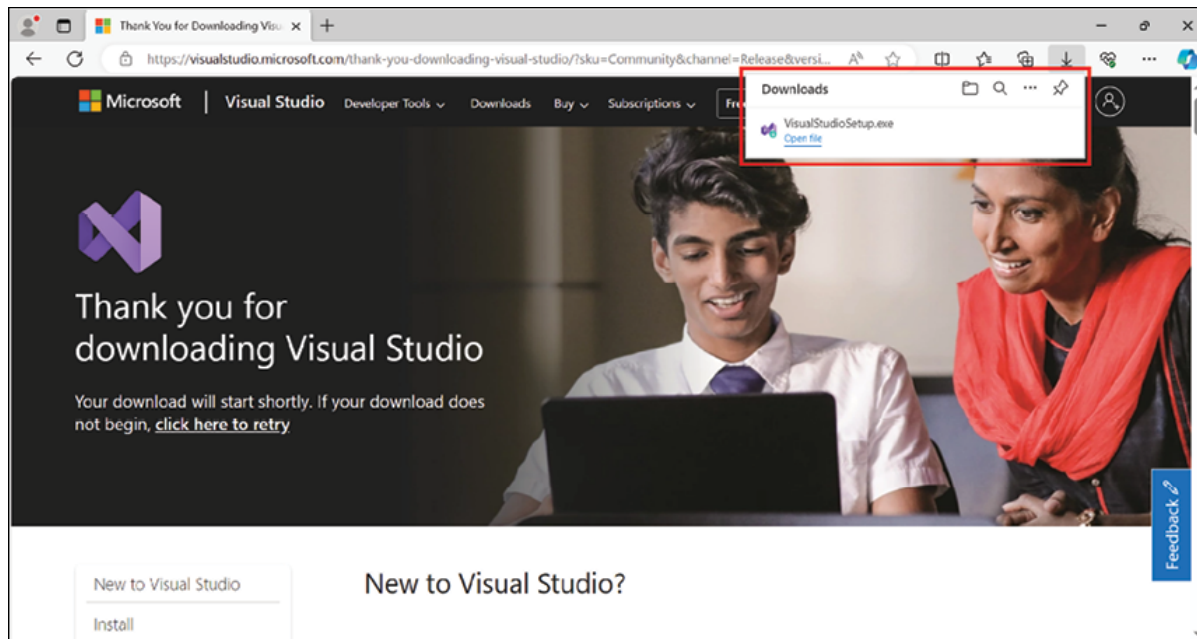


Figure 3.2: Check the Downloads folder for the setup file

2. After installing the Visual Studio installer, you can run it to customize your installation, as shown in the following figure:

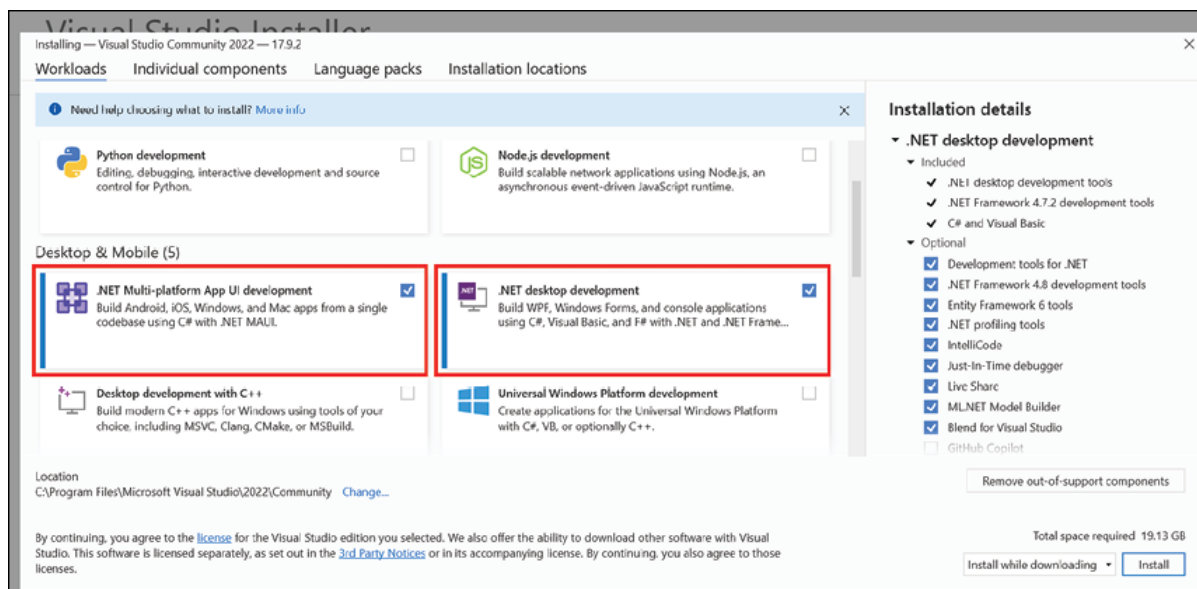


Figure 3.3: The Visual Studio installer

3. Choose the workloads indicated and the optional components which are selected by default and click on **Install**.
4. The installation process is fully automated and you can check the status screens for the progress of the installation. Refer to the following figure for the status message when installation is successfully completed:

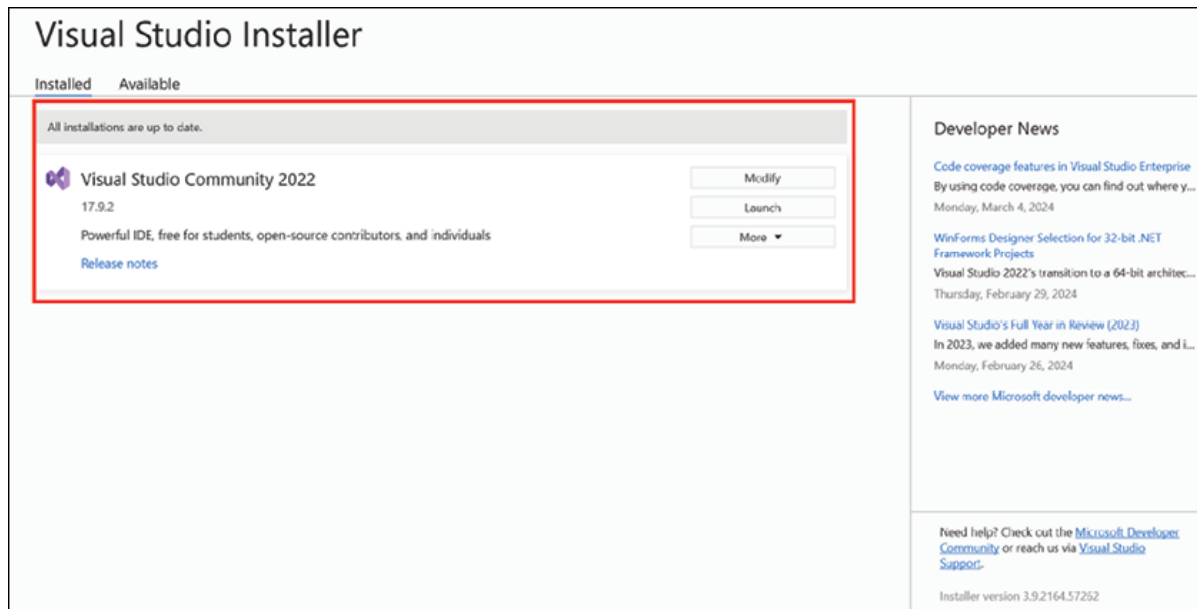


Figure 3.4: Completed status of Visual Studio installation

In case you have already installed Visual Studio IDE for other purposes, you need not uninstall it.

You can modify the existing installation and add the required workloads or components any time. You can also install multiple versions of Visual Studio side by side.

Testing the environment setup

Once your Visual Studio installation is completed, follow these steps:

1. Click the **Launch** button to start the Visual Studio IDE.
2. You can create a free account to preserve your personalized preferences across devices, but for the purpose of this chapter, you can skip the step for now. Refer to the following figure for an illustration of the sign up screen:

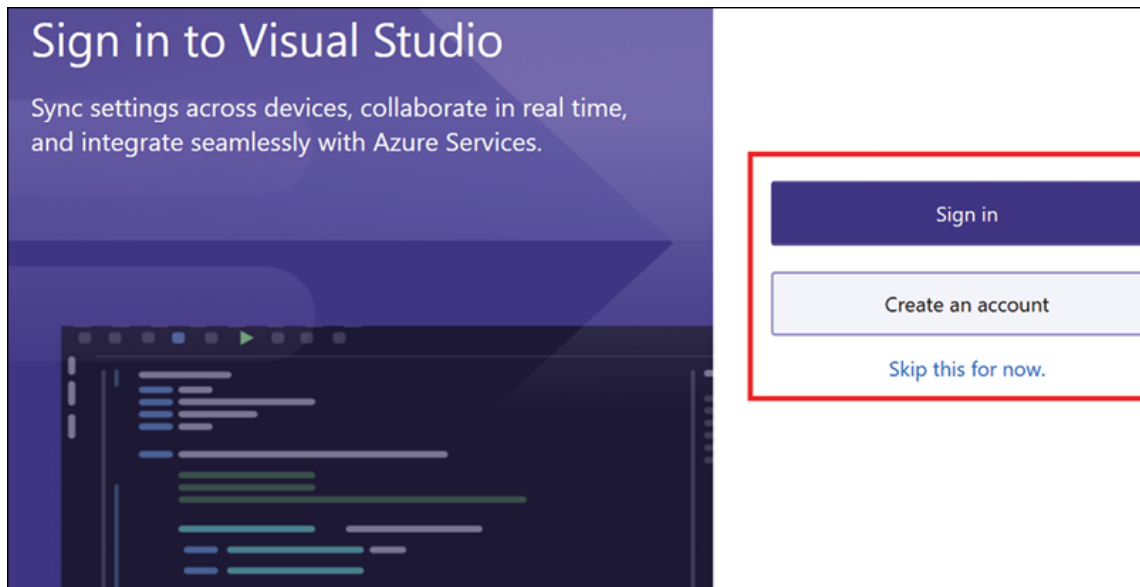


Figure 3.5: Sign-in screen

3. Assuming that you are a first-time user, it is always good to start with a simple C# Console application to get familiar with Visual Studio IDE.
4. In the start screen, select **Create a new project**. Refer to the following figure for the option to select the create a new project:

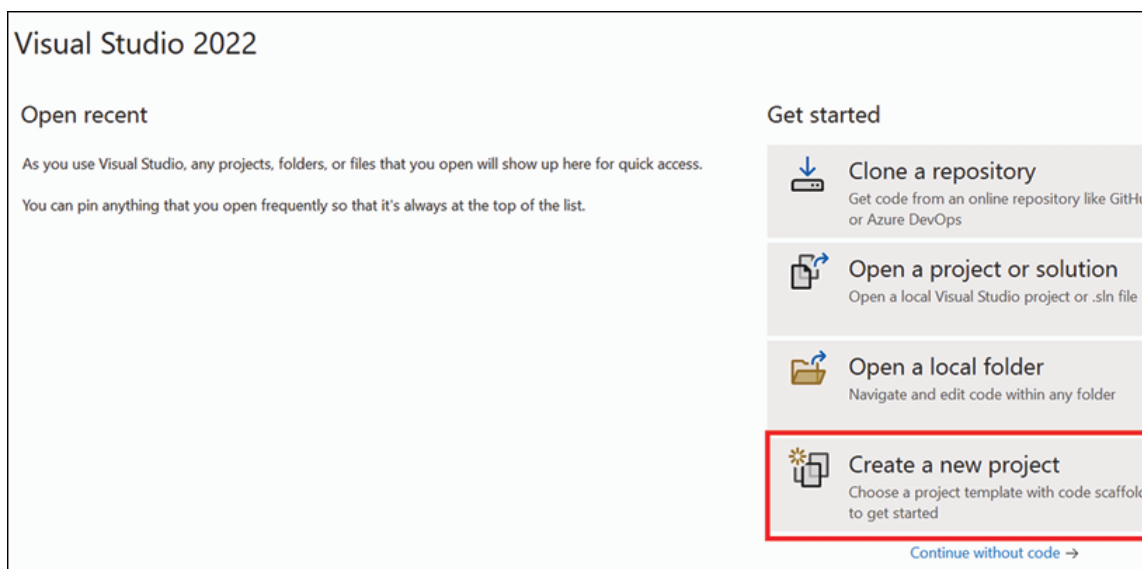


Figure 3.6: Creating a new project

5. You can scroll through the list of templates or use the template search box to type in Console application. Click on the **Console App** template as shown in the figure:

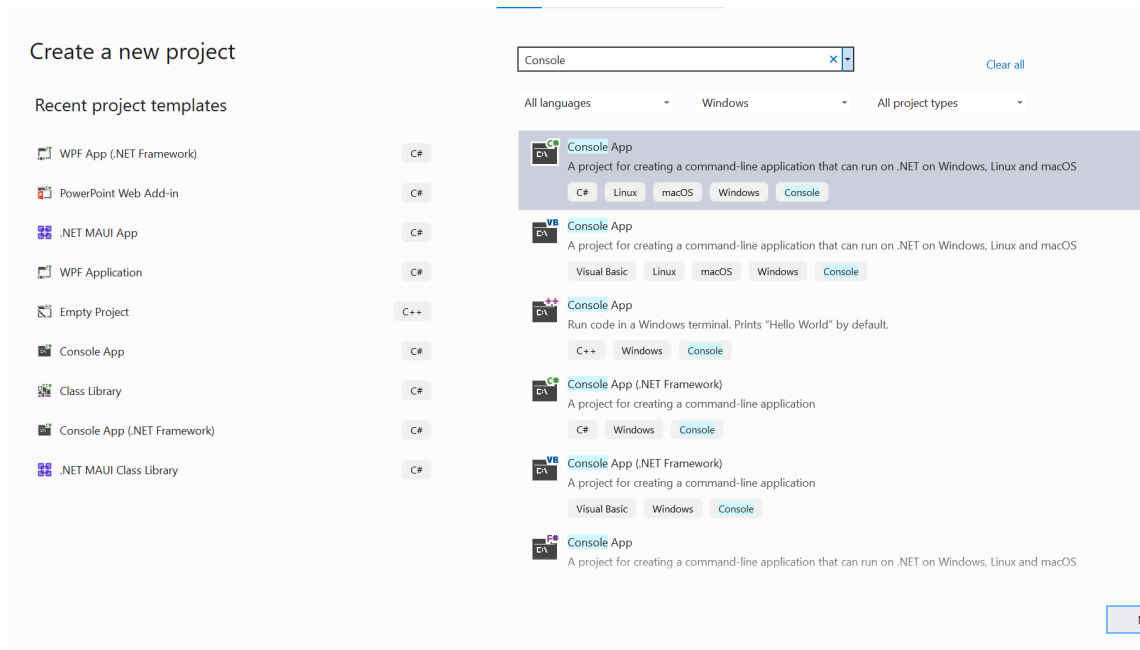


Figure 3.7: Selecting the Console App template

6. The programming language that we will be using in this book is C# and so ensure you select C# in the language drop-down menu. Refer to the following figure for the option to select the programming language:

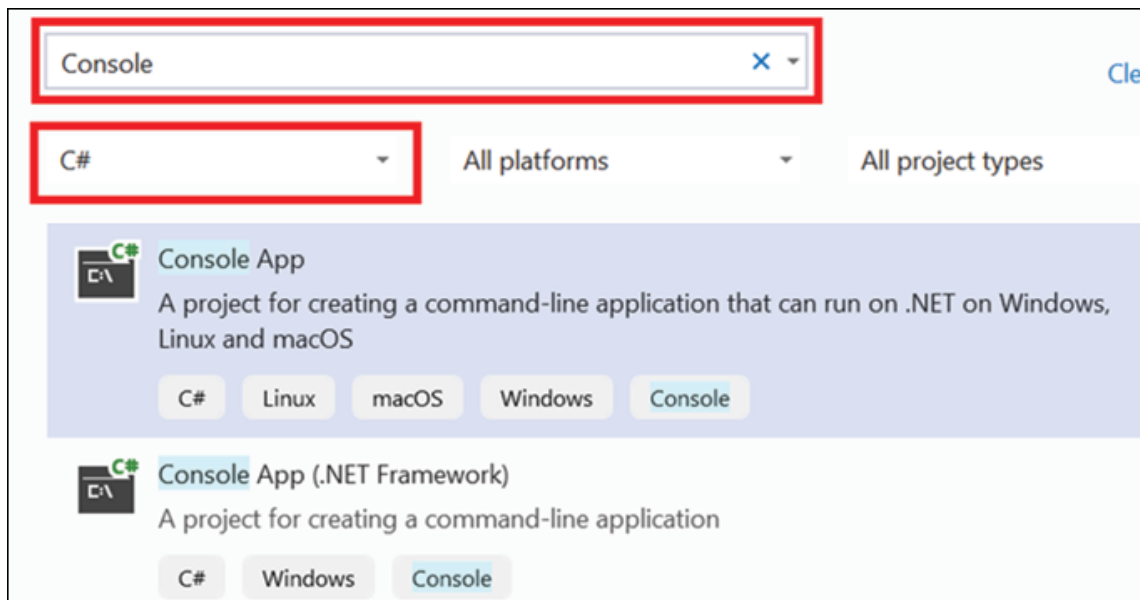


Figure 3.8: Selecting the programming language

7. Select the **Console App** template and click **Next**.
8. In the next screen, you will be able to configure your new project. Refer to the following figure:

Configure your new project

Console App C# Linux macOS Windows Console

Project name

ConsoleApp1

Location

C:\Users\admin\source\repos

Solution

Create new solution

Solution name ⓘ

ConsoleApp1

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1"

Figure 3.9: *Configuring the new project*

9. Provide a name for the project. It is advisable to use alphanumeric characters only without any space or special characters. Refer to the following figure for selecting the project name:

Configure your new project

Console App C# Linux macOS Windows Console

Project name

ConsoleApp1

Location

C:\Users\admin\source\repos

Solution

Create new solution

Solution name ⓘ

ConsoleApp1

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1\"

Figure 3.10: Selecting the project name

10. Choose the desired location. As you can see, by default, the project is saved it in the **\source\repos** folder under the logged in user's profile. Refer to the following figure:

Configure your new project

Console App C# Linux macOS Windows Console

Project name

ConsoleApp1

Location

C:\Users\admin\source\repos

Solution

Create new solution

Solution name ⓘ

ConsoleApp1

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1\"

Figure 3.11: *Selecting the location for the project files*

11. A solution in Visual Studio is a container to hold multiple projects. You can choose a new solution or add the project to an existing location. We will choose to create a new solution, which will have the same name as that of the project by default. If desired, you can change it, but this is purely optional. Refer to the following figure:

Configure your new project

Console App C# Linux macOS Windows Console

Project name

ConsoleApp1

Location

C:\Users\admin\source\repos

Solution

Create new solution

Create new solution

Add to solution

ConsoleApp1

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1"

Figure 3.12: *Configuring the solution*

12. Ignore the option to place solution and project in the same directory and leave it at its default unchecked setting. Refer to the following figure to select the option to create a new solution:

Configure your new project

Console App C# Linux macOS Windows Console

Project name

Location

Solution

Solution name ⓘ

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1\"

Figure 3.13: Creating a new solution

13. Click **Next**. Under **Additional information**, you can choose the **Framework** version that you want to target but leave it at the default setting for now. Refer to the following figure:

Additional information

Console App C# Linux macOS Windows Console

Framework ⓘ

☒ Do not use top-level statements ⓘ

☐ Enable native AOT publish ⓘ

Whether to generate an explicit Program class and Main method instead of top-statements.

Figure 3.14: Configuring additional information for the project

14. Also check the option to avoid using top-level statements, which is a new feature introduced in a recent version of C# to directly write executable code without wrapping the code in a C# class or method. Click **Create**.

15. By default, a simple Console application is created for you. We will discuss C# in more detail in a subsequent chapter, but in this chapter, we will just focus on the basic features of the IDE and the development process to get you started.
16. The key sections of the IDE are highlighted in the following figure:

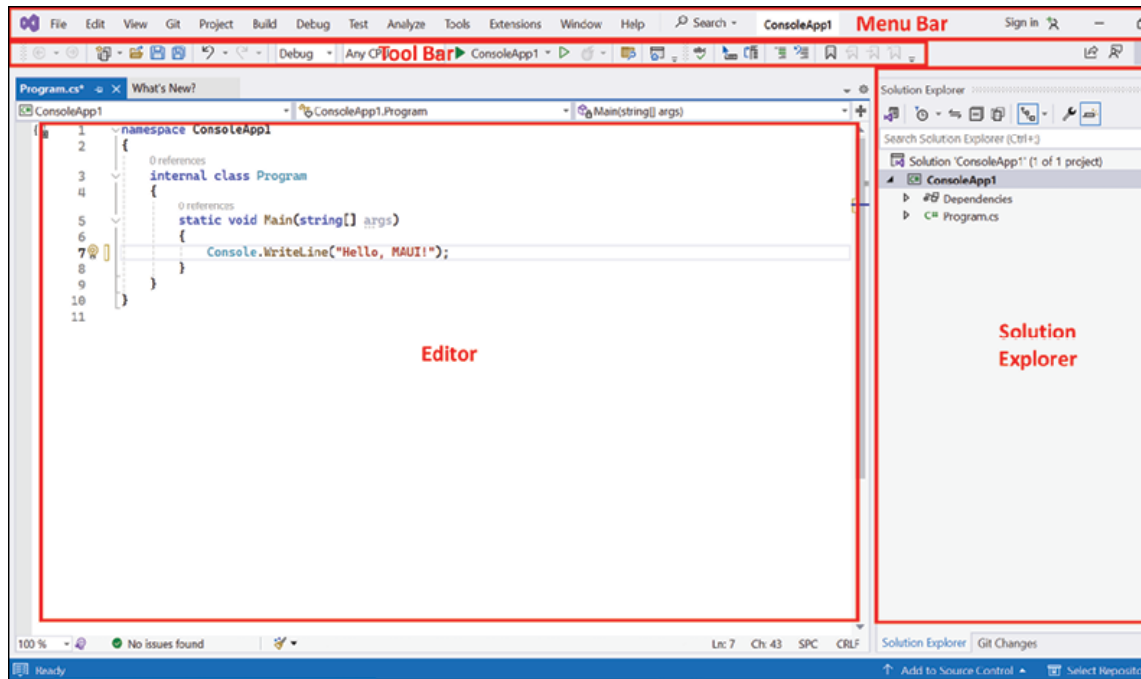


Figure 3.15: Visual Studio IDE Layout

17. In the **Solution Explorer**, you will find a **Program.cs** which contains the program.
18. Double-click the file to open it in the editor area to make any changes. For this example, we have just updated the string in line no. 7 to display a simple message to the user when the program is executed. Refer to the following figure:

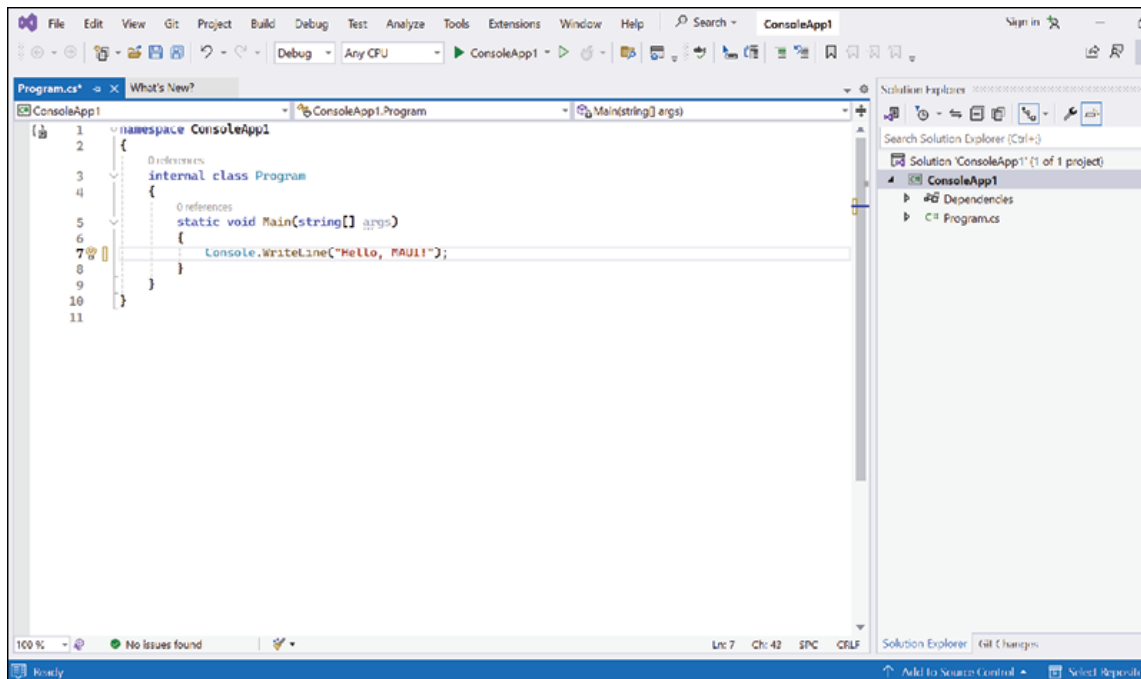


Figure 3.16: Editing in Visual Studio

19. After editing the program, it has to be compiled and then executed. Click on **Debug | Start Debugging** in the menu bar. An output pane will appear at the bottom of the editor to display the status of the compilation and execution process. Refer to the following figure:

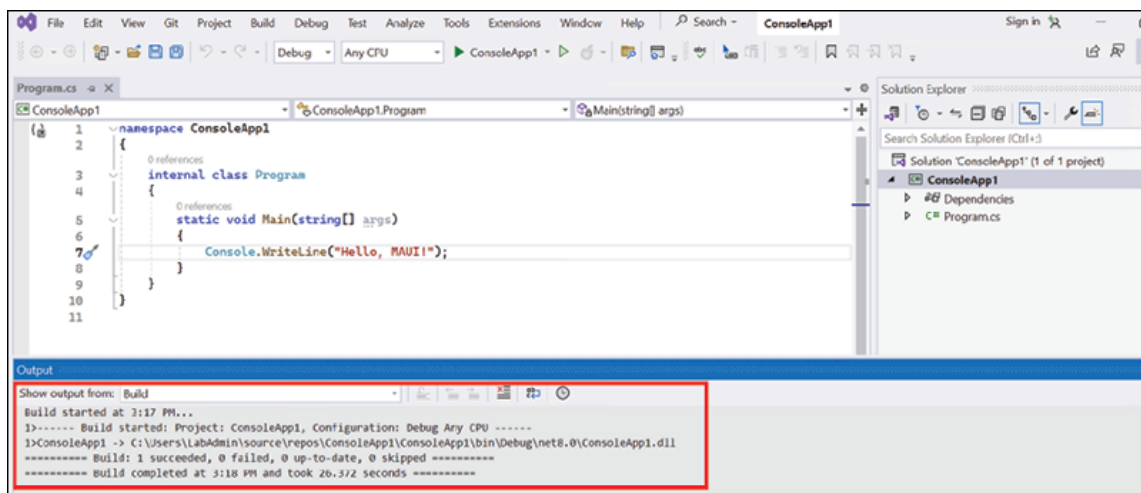
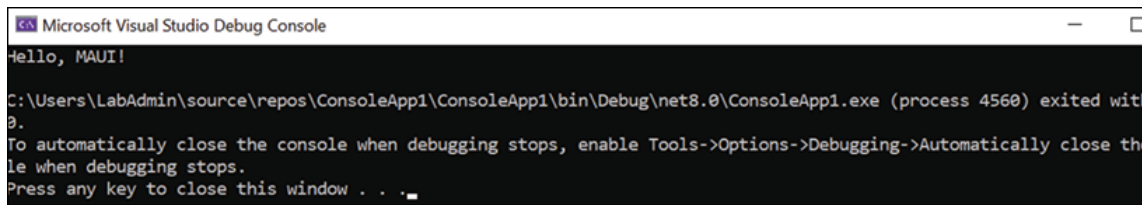


Figure 3.17: Building the program

20. If the program has no errors, it will be successfully compiled and you should be able to see the output of the program in a console window:



```
Microsoft Visual Studio Debug Console
Hello, MAUI!
C:\Users\LabAdmin\source\repos\ConsoleApp1\ConsoleApp1\bin\Debug\net8.0\ConsoleApp1.exe (process 4560) exited with
0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close th
le when debugging stops.
Press any key to close this window . . .
```

Figure 3.18: The output of the program

21. In case there is an error in the program, as shown in the following figure where we intentionally removed the semi-colon at the end of line no. 7, the compiler will display an error during the compilation process, and the program will not be executed:

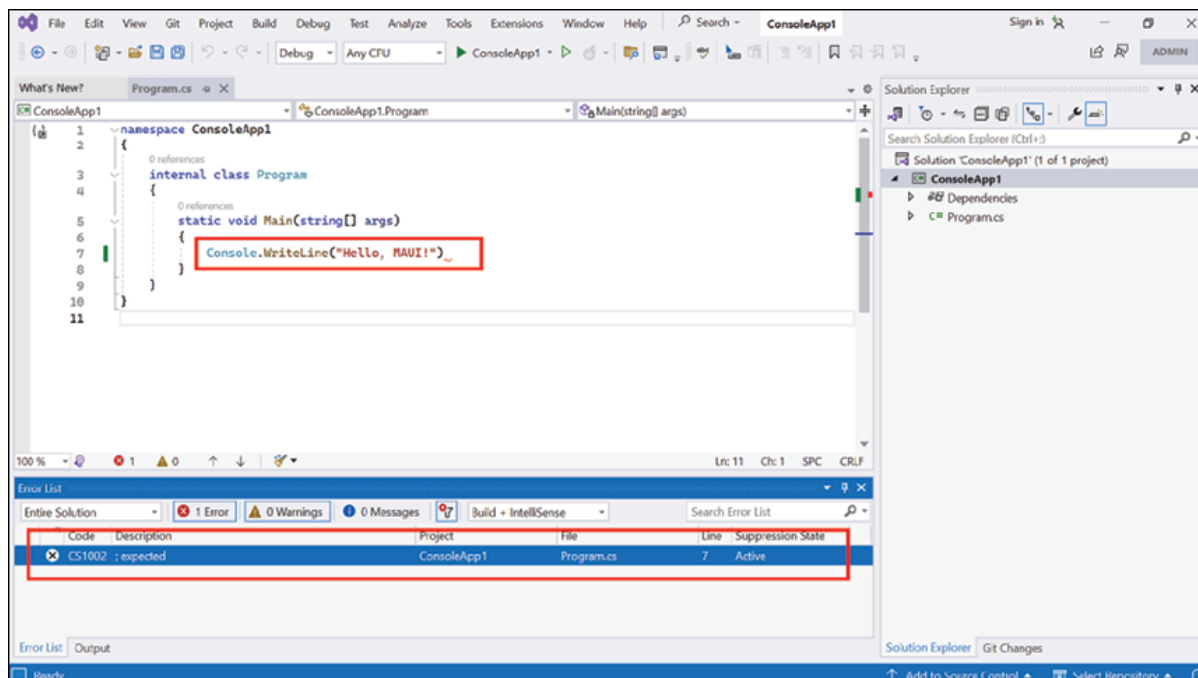
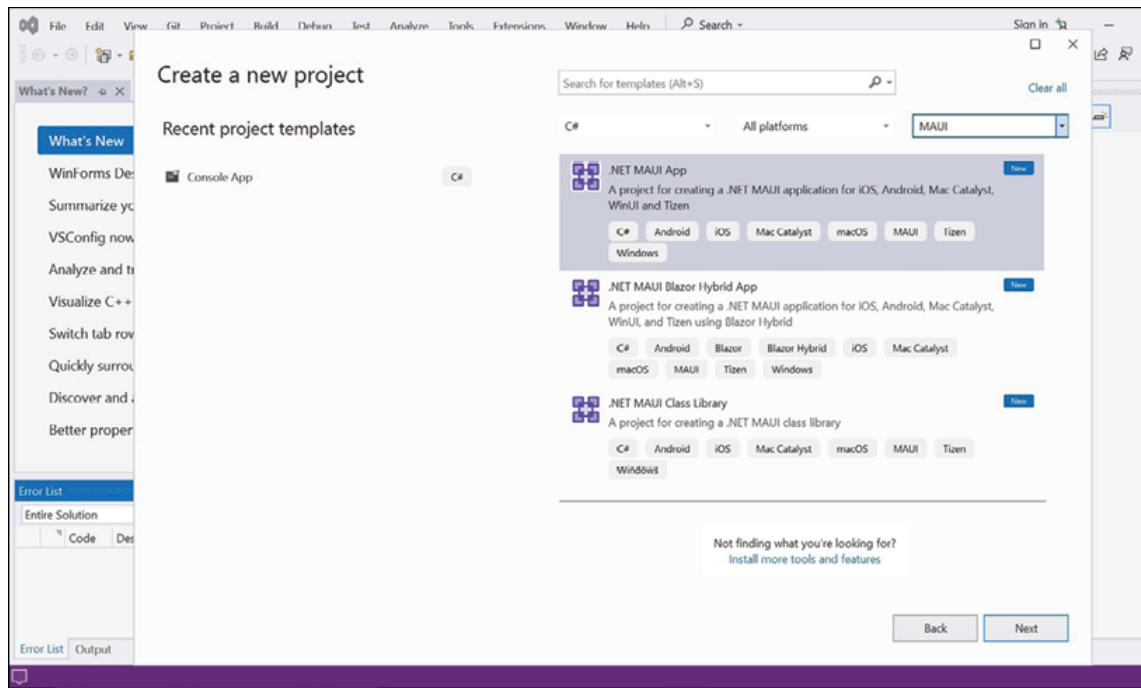


Figure 3.19: Viewing errors in Visual Studio IDE

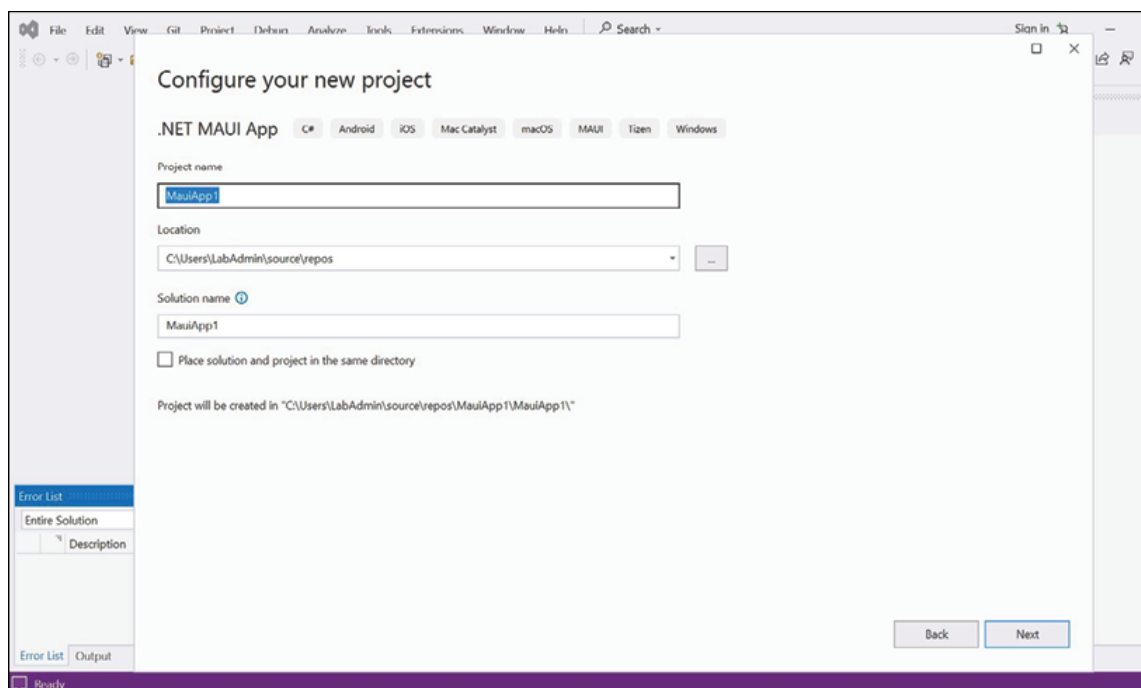
Debugging on Windows

Now, that you are comfortable with the core functionalities of Visual Studio, it is time to write a simple NET MAUI App and test it on a Windows machine. Follow these steps for debugging on Windows:

1. From the menu bar, select **New Project**.
2. Search for MAUI and select **.NET MAUI App**:



3. Enter a suitable **Project Name** and choose the **Target Framework** version just like you did for the console application and create the project:



4. A boilerplate solution is already created for you so you can directly execute this application:

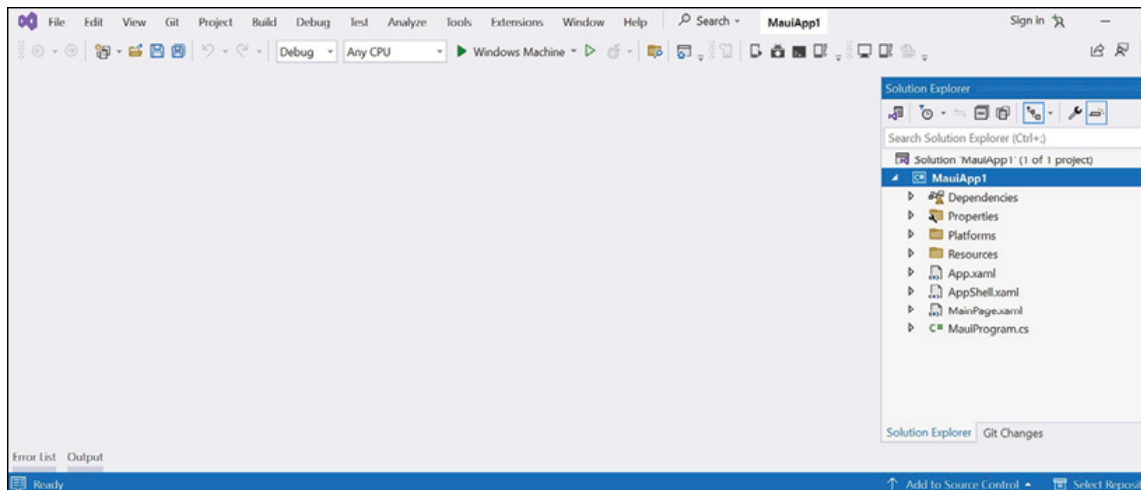


Figure 3.22: The default .NET MAUI project structure

5. From the toolbar click on the button which displays **Windows Machine** to start debugging on Windows device. Refer to the following figure:

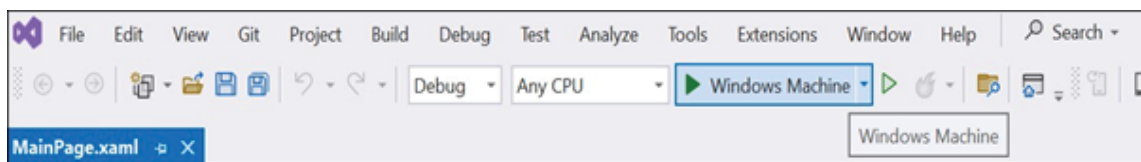


Figure 3.23: Debugging on Windows Machine

6. If this is the first time you are trying to run a .NET MAUI application on your Windows PC, you will see the dialog as shown in the below figure, prompting you to enable Developer Mode for Windows.

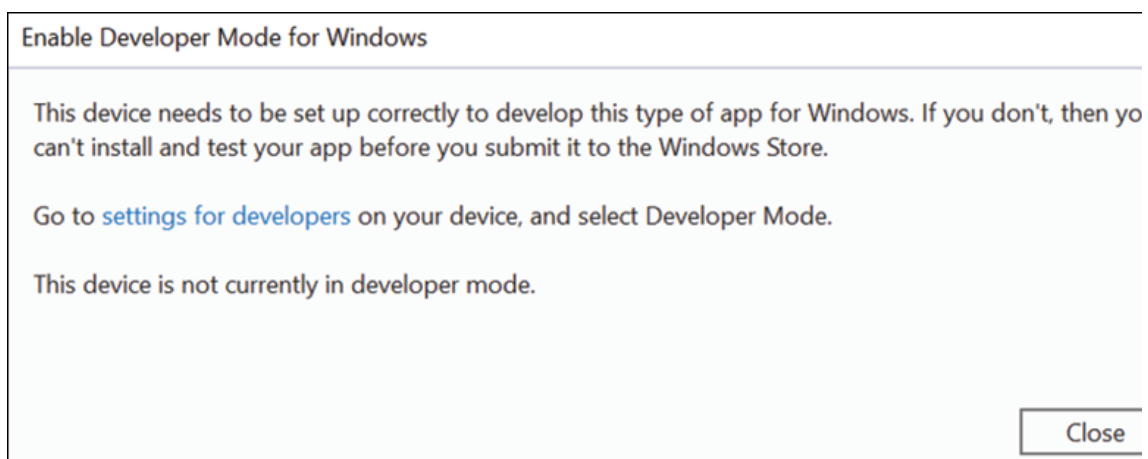


Figure 3.24: Dialog to enable Developer Mode for Windows

7. To enable **Developer Mode**, you can either click on the hyperlink provided in the dialog or alternatively access **Developer** settings from the **Settings** menu:

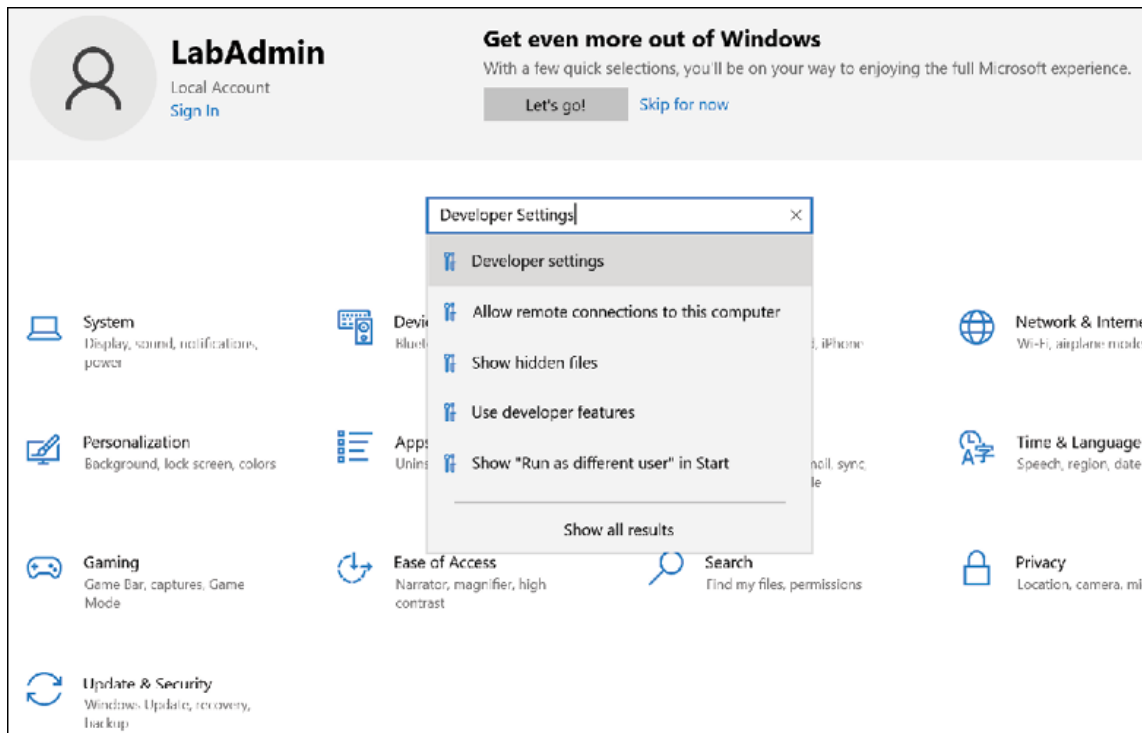


Figure 3.25: Searching for Developer Settings

8. In the menu **For developers**, enable the option under **Developer Mode** to **Install apps from any source, including loose files**:

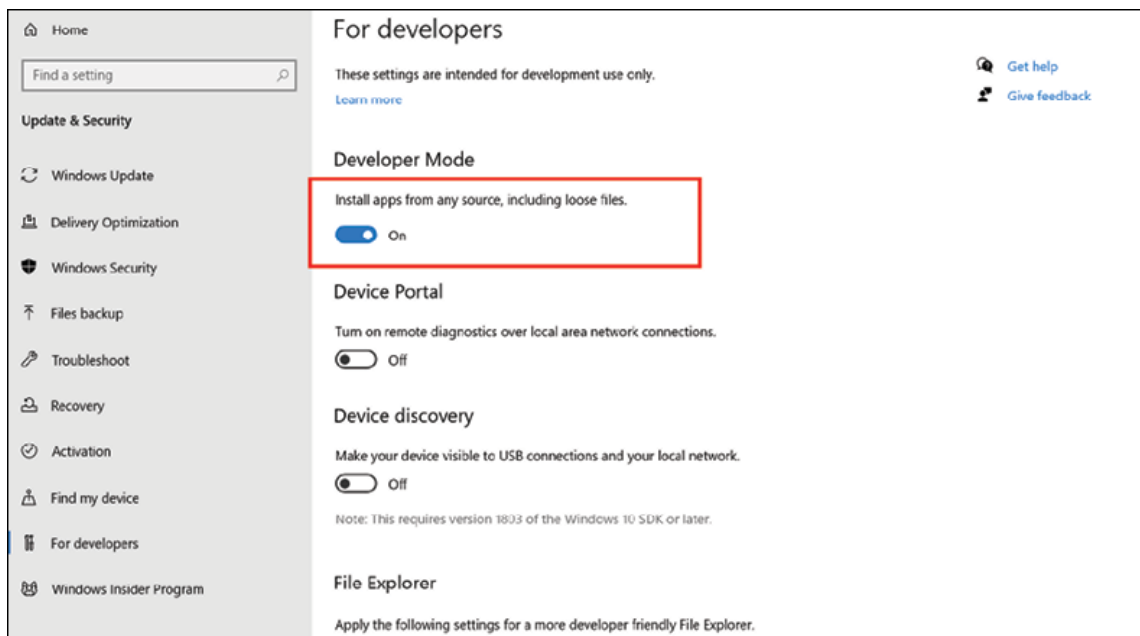


Figure 3.26: Enabling Developer Mode

9. Confirm to turn on the **Developer Mode**:

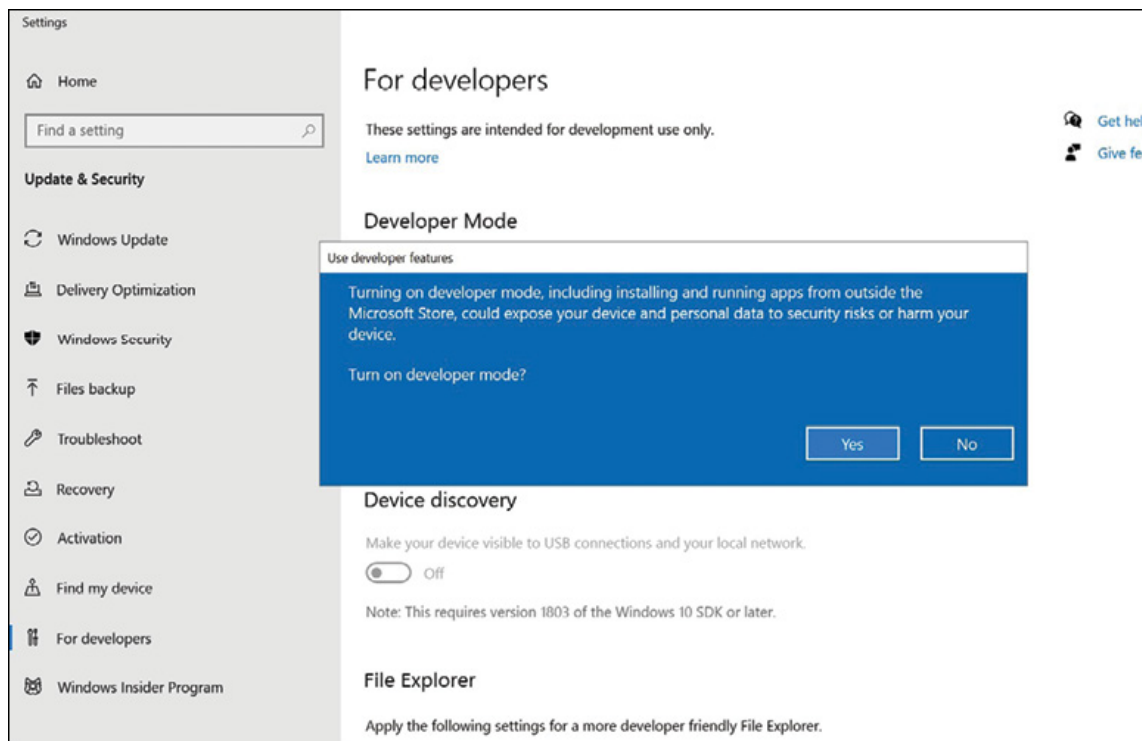


Figure 3.27: Confirmation to enable developer mode

10. Now, the application will be compiled, and you should be able to view the output of the default application created by Visual Studio:

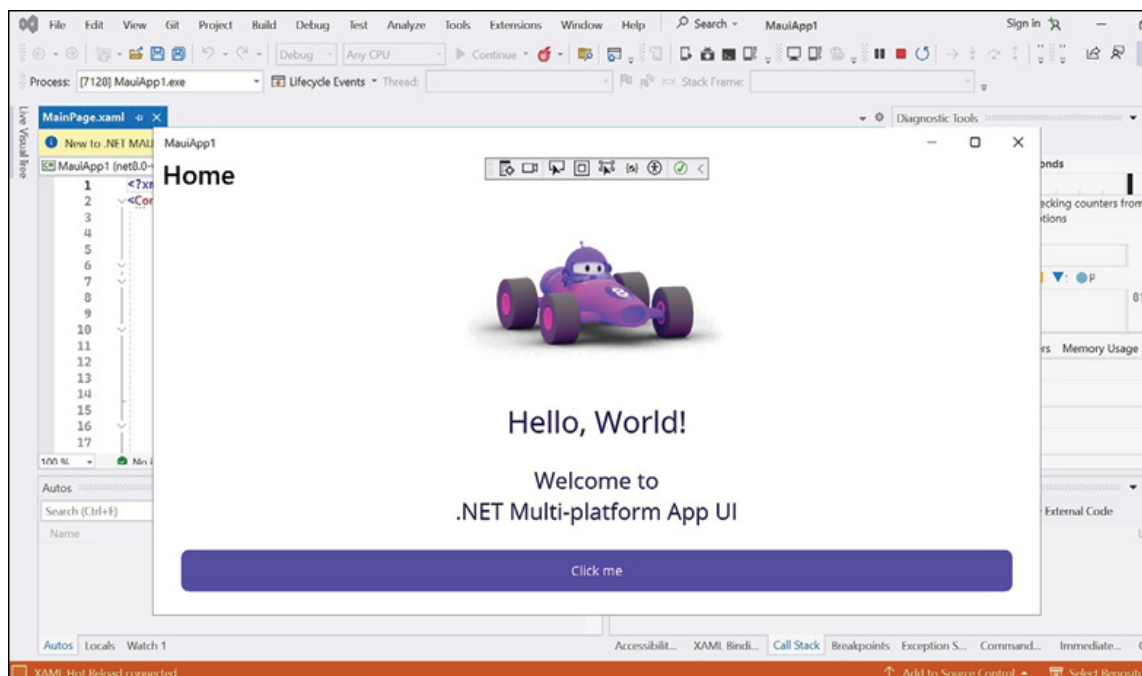


Figure 3.28: Output of .NET MAUI executing in Windows Machine

11. Click on the button a few times and observe that the click count is incremented in the button's text.

Your PC is now enabled for developing and debugging on Windows.

Debugging on Android Emulator

Visual Studio lets you run the app in emulators that can simulate different Android devices. Follow these steps to debug on Android Emulator:

1. First, you need to create and configure the emulator. Go to **Tools... Android... Android Device Manager**:

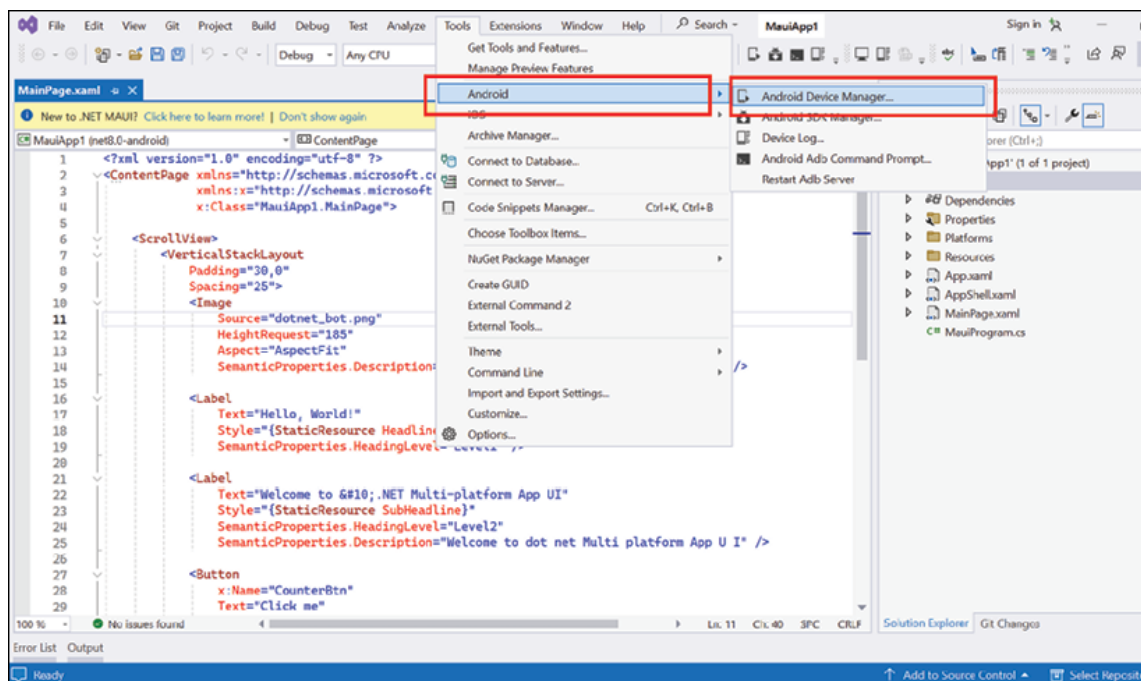


Figure 3.29: Selecting Android Device Manager

2. Initially, the **Android Device Manager** will be empty. You can add a new device by clicking on the **New** button:

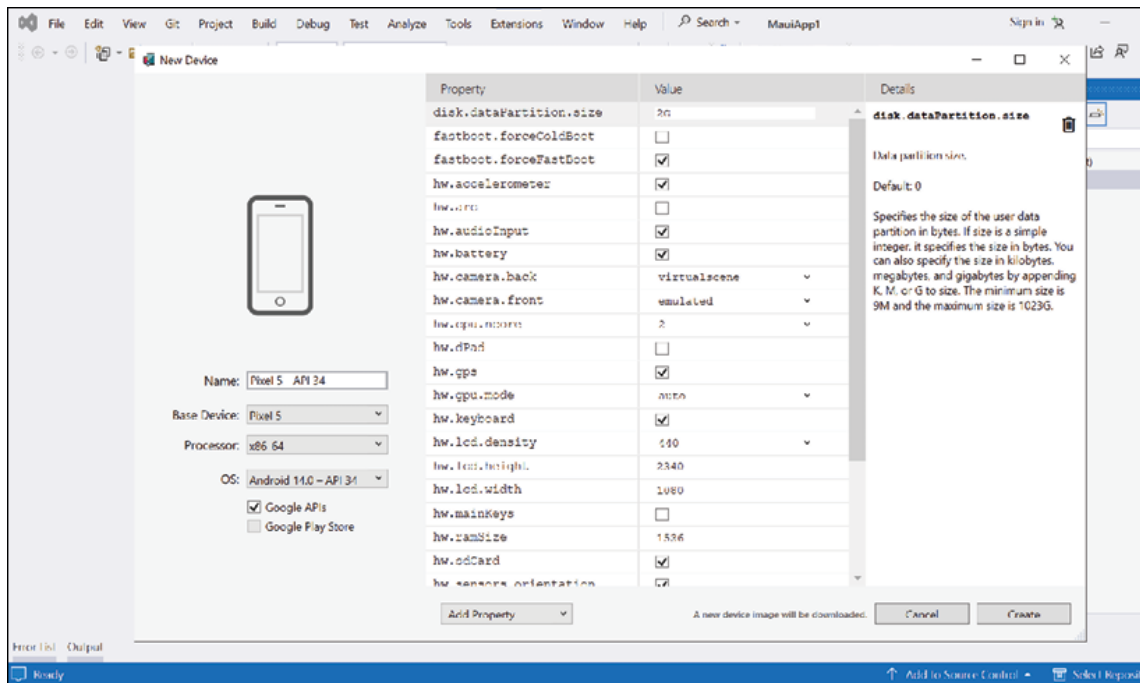


Figure 3.30: Adding a new device in Android Device Manager

3. You can leave the default options as it is and click on **Create**:
4. Accept the license terms that are displayed.

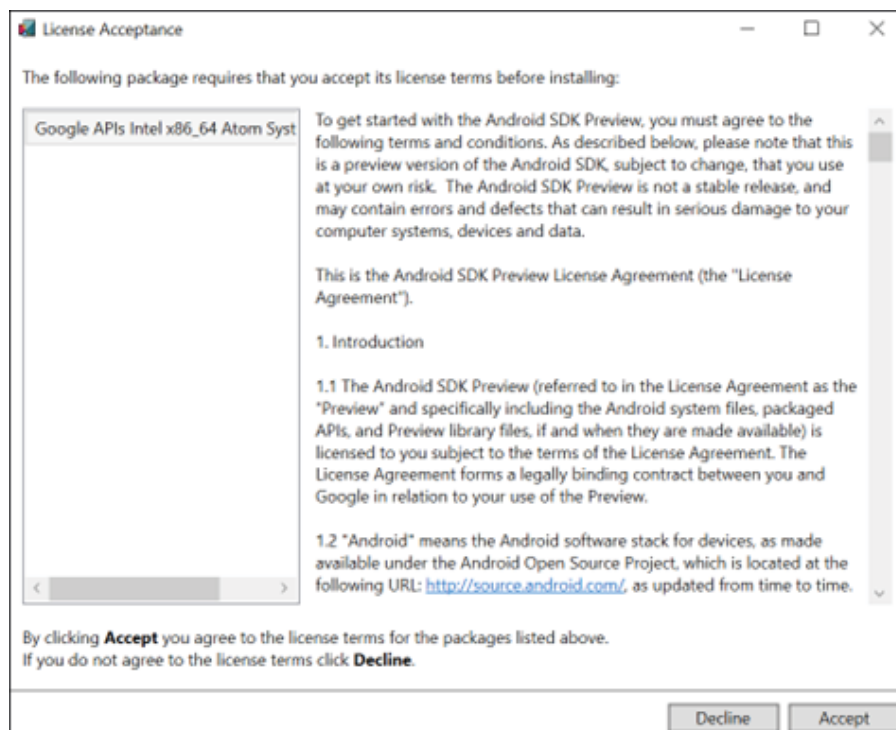


Figure 3.31: License Agreement for installing Android virtual device

5. Visual Studio will then download the device image and install it. After this process is completed, you should be able to see the virtual device listed in the Device Manager:

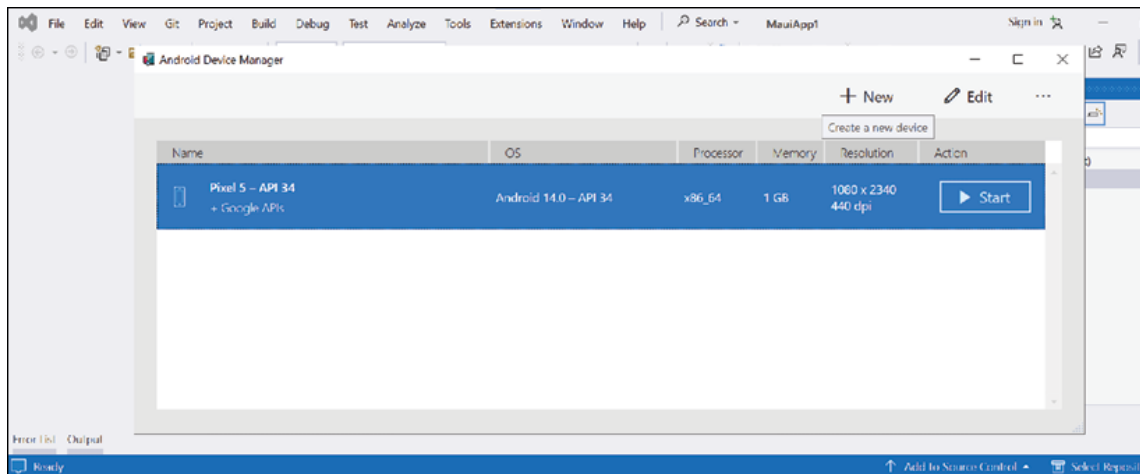


Figure 3.32: Virtual device installed

6. The virtual device will now be visible to select in the drop-down list to select the target for debugging:

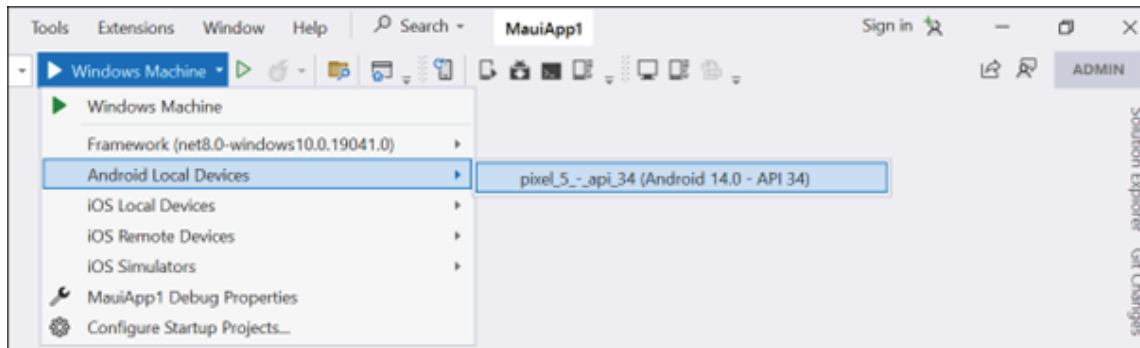


Figure 3.33: Selecting the Android Virtual Device for debugging

7. Select the virtual device and click on the **Play** button.
8. The emulator will start and if the code is compiled without any errors, the app will be launched inside the emulator.

Debugging on Android devices

While the Android Emulator is useful for testing your app quickly, it is important to also test it on a real Android device. To do this, you will need to turn on *Developer Mode* on your device and connect it to your computer.

To enable developer mode, follow these steps:

1. Go to the **Settings** screen.
2. Select **About phone**.

3. Tap on **Build Number** seven times until you see the message **You are now a developer!**

Refer to the following figure:

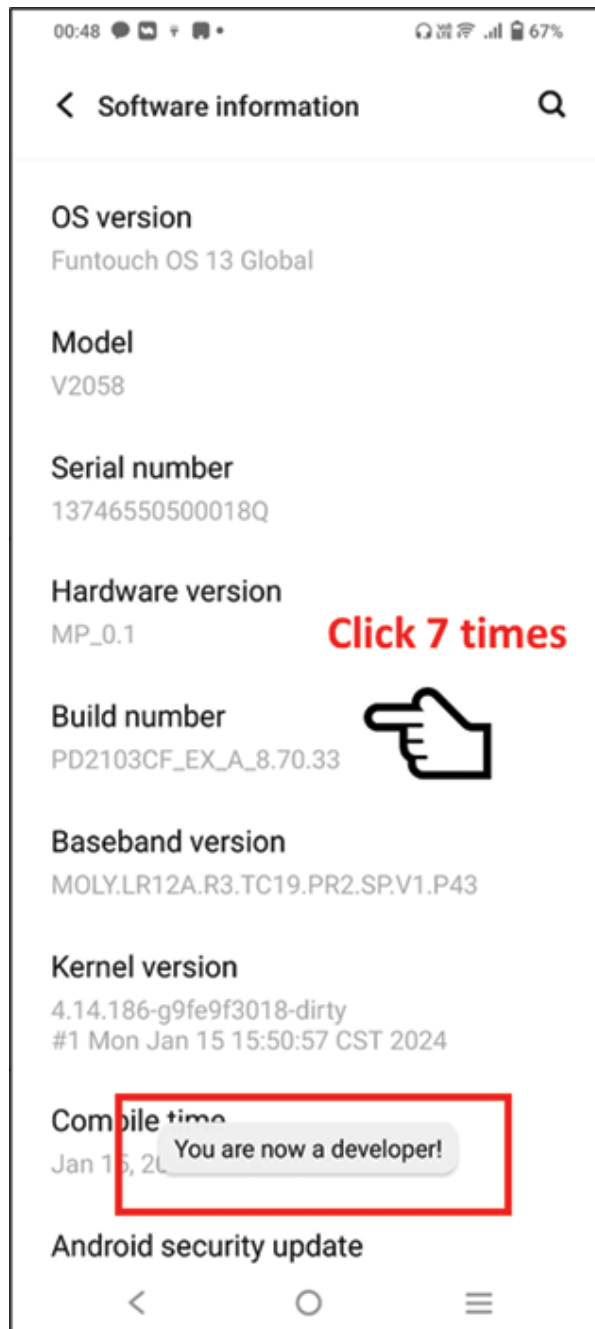


Figure 3.34: Turning on Developer Mode on Android device

Once Developer Mode is enabled, you need to turn on USB debugging:

1. Go to the **Settings** screen.
2. Select **Developer options**.

3. Turn on the **USB debugging** option.
Refer to the following figure:

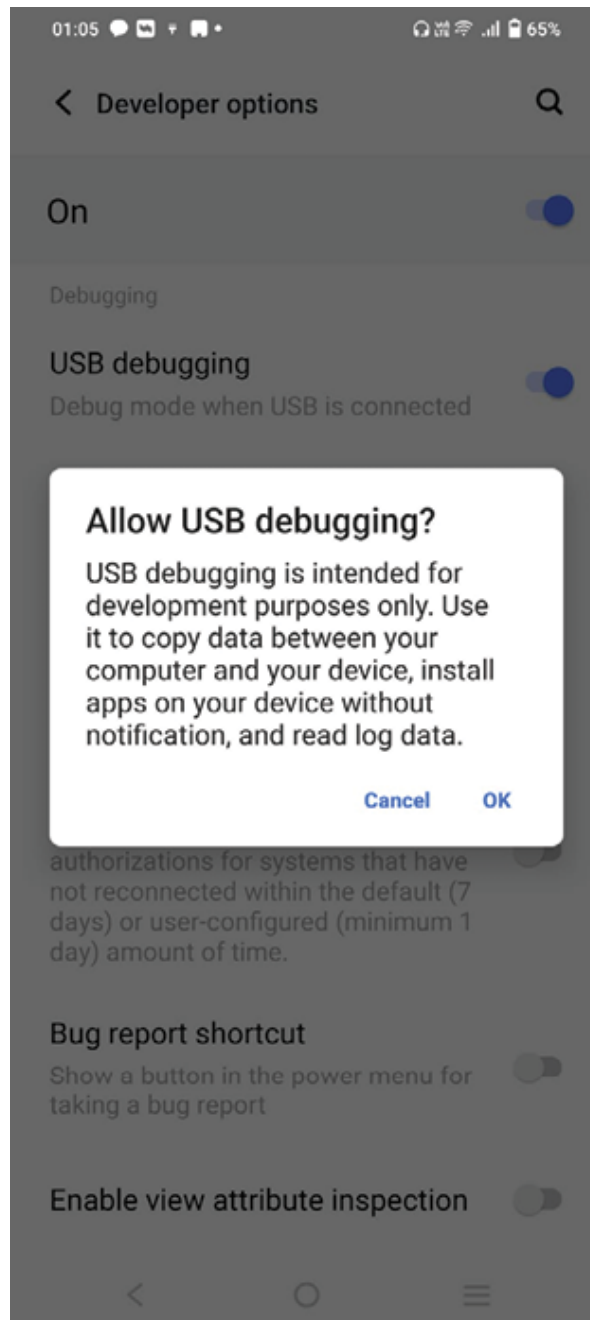


Figure 3.35: Allow USB debugging

4. Now, connect your device to the computer using a USB cable. If it is your first time connecting for debugging, you will see a prompt on your device asking if you trust the computer.
5. Select the option to always trust this computer to avoid seeing the prompt again.

6. Once the device is connected, it will be visible for you to select in the drop-down list of target devices.
7. Select it and click the **Play** button to execute the app on your device.

The procedure explained here is good for side-loading onto a device for testing purposes.

For distributing the app to a large number of users, it is recommended to create a Developer account in Android Play Store and publish the app to the Play Store. The procedure is outside the scope of this book.

iOS device provisioning

As already mentioned at the beginning of this chapter, we will discuss setting up the development environment in Windows PC. If you have a Mac and are developing on a Mac for the iPhone, you can follow the instructions given here.

When developing a .NET MAUI app for iOS on a Mac, you will need to install the .NET runtime and the latest version of Xcode from the App Store. Once this is done, you can execute simple commands in the Terminal to build and deploy the app to a device or to a simulator.

To test it on an iOS simulator, you must provide its **unique device ID (UDID)** in the command while building the app in the Terminal. The UDID will be available from within the **Simulators** tab in the Xcode setting for Devices and Simulators.

To test it on a real iOS device, you need to follow some steps, which are referred to as device provisioning. Firstly, you need to have an active subscription to Apple's Developer Program—which may be either the regular one for individuals or the Enterprise program meant for organizations. Developers need to create a certificate, register the device they are using for development, and create an App ID and a provisioning profile.

When you deploy the app to a device, a provisioning profile is also installed on the device. This profile checks if the app was signed correctly and if it is allowed to run on that device. It checks things like the certificate used to sign the app, the App ID and if the device is registered in the provisioning profile.

Pair to Mac for iOS development

In case you prefer to use Windows to develop iOS apps, it is possible to do so if you can connect to a Mac in the same network as Apple's Build tools run only on a Mac. Visual Studio provided a Pair to Mac feature using which you can discover, connect to and authenticate your Mac build host. With this approach, multiple users can connect to the same Mac simultaneously.

The steps to be followed to establish the pairing are demonstrated in the following figures:

1. On your Mac, invoke Spotlight by pressing *Cmd + Space*, and search for **Remote Login** and open the **Sharing System Preferences** window:

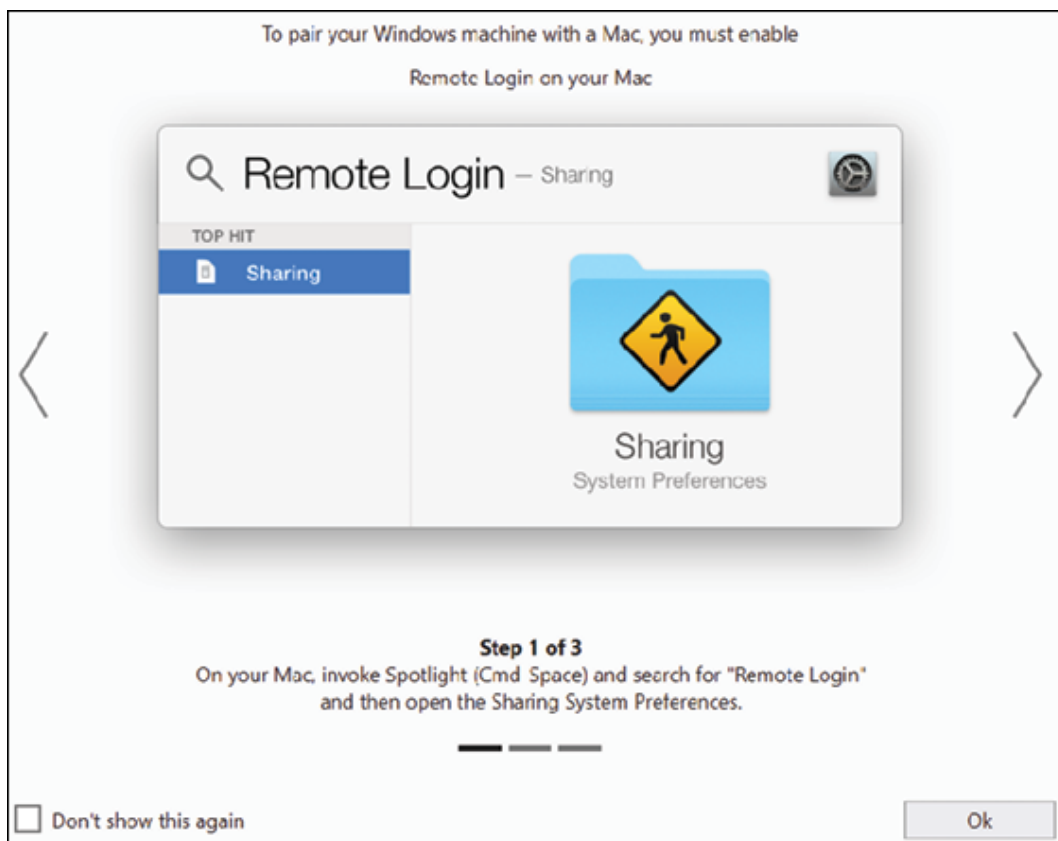


Figure 3.36: Accessing Sharing System Preferences

2. Enable the **Remote Login** option in the Sharing System Preferences window:

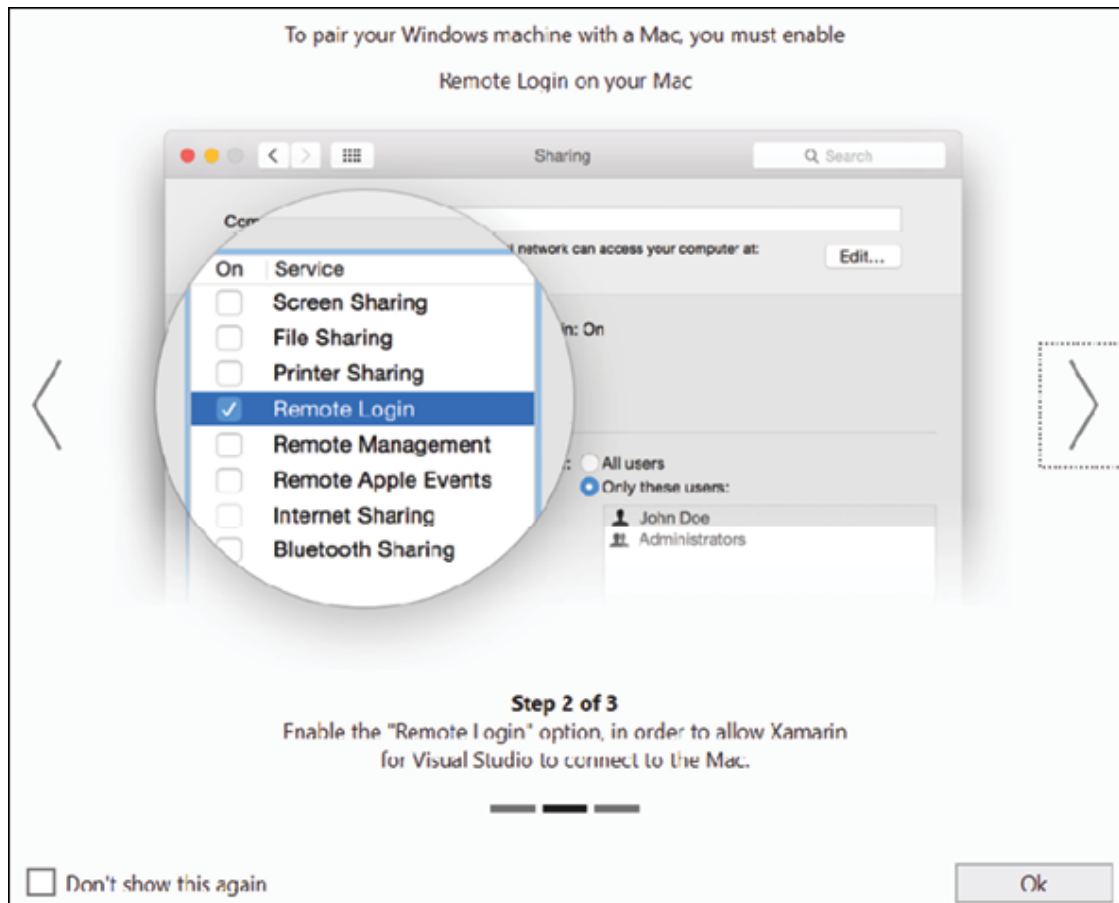


Figure 3.37: Remote Login option in Sharing System Preferences

- Set access for **Only these users** and add your username in the list so that you can access the Mac through Remote Desktop:



Figure 3.38: Enabling Remote Login

If you have successfully paired your Visual Studio environment to a Mac build host, you can test your app in an iOS simulator running on your Windows PC.

Conclusion

In this chapter, you have learned how to set up your development environment to build .NET MAUI apps. We also discussed how you can run your apps in multiple environments, including Windows, Android, and iOS. In the next chapter, we will look into C#, the programming language that will be used for the app development, that will equip you with the essential knowledge needed for the content in the upcoming chapters of the book.

Points to remember

Here are some key takeaways from this chapter:

- Install Visual Studio 2022 Community Edition.
- Enable Developer Mode in your Windows PC to test your apps in Windows.
- Install and configure Android emulator to test your apps in Android Virtual device.
- You can also enable developer mode in the Android device and run the app in your device.
- Testing apps on iOS requires you to have a Mac.
- You can develop on Windows if you can pair a Mac build host to your Visual Studio environment.
- Distribution of apps requires you to have a Windows Marketplace, Play Store or App Store account.

CHAPTER 4

A Crash Course in C#

Introduction

This chapter gives a concise and focused introduction to C# designed to equip beginners with the essential knowledge needed to follow the chapters ahead in this book. While this chapter does not provide an exhaustive coverage of the C# language, it serves as a vital primer to ensure that readers, especially those new to C# programming, can comfortably follow and comprehend subsequent discussions. For those seeking in-depth exploration, this publisher offers comprehensive books dedicated to the topic in detail.

Structure

In this chapter, we will discuss the following topics:

- Introduction to C#
- Types, variables, and constants
- Strings
- Operators and expressions
- Branches and loops
- Classes, methods and properties
- Interfaces and inheritance

- Delegates and events
- Arrays and lists

Objectives

After reading this chapter, you will be able to understand:

- Basic programming concepts in C#
- Types, variables, and constants
- Manipulate and work with string data type
- Use operators and build expressions
- Implement branching and decision-making statements
- Use object-oriented programming features for modular code
- Explore interface implementation and use the concept of inheritance
- Leverage the power of delegates and events
- Work with arrays and lists for efficient storage and retrieval of data

Introduction to C#

C# (pronounced C-Sharp) is like a *Swiss Army* knife of programming languages. It is powerful, yet easy to learn and versatile to develop all kinds of applications, from websites to mobile apps to games and everything in between. It was developed at Microsoft by *Anders Hejlsberg*, and introduced with the .NET Framework in 2002 and has been continuously evolving since then. Presently, it is in Version 12.0. The core syntax is like other C-style languages like C, C++, and Java and follows similar rules:

- A program is composed of statements written in separate lines.
- Semicolons are used to denote the end of each statement.
- Curly brackets are used to group statements, either into blocks, conditional and looping statement blocks, methods (functions), classes, etc.
- Comments may be single line using `//` or multi-line using `/* ... */`

We will now examine specific features of the C# programming language that will help you get started with app development using .NET MAUI.

Types, variables, and constants

Variables are containers used to store and label information or data that you can use in your program. When creating a variable, you need to define the data type of the variable as well. Refer to the following code for declaring a variable:

```
1. int x = 5; // declaring a variable of type integer and  
initializing with the value 5
```

There are different data types that you can declare in C#, and we will discuss those in a while, but what is important to know here is that C# is a strongly typed language. This means that once a variable is declared with a certain data type, it cannot be changed during the lifetime of the variable.

Constants, similar to variables, are also containers to store data, but the difference is that the value, once set, does not change throughout the program:

```
1. const int y = 7;
```

Let us now look at data types in C#. In any programming language, data types help the programmer specify the following information:

- The kind of data to store (numeric, non-numeric, etc.).
- Range of values that can be stored.
- Amount of memory used to store the data.
- Type of operations that can be performed.

To understand the different types of data types in C#, let us refer to the following figure:

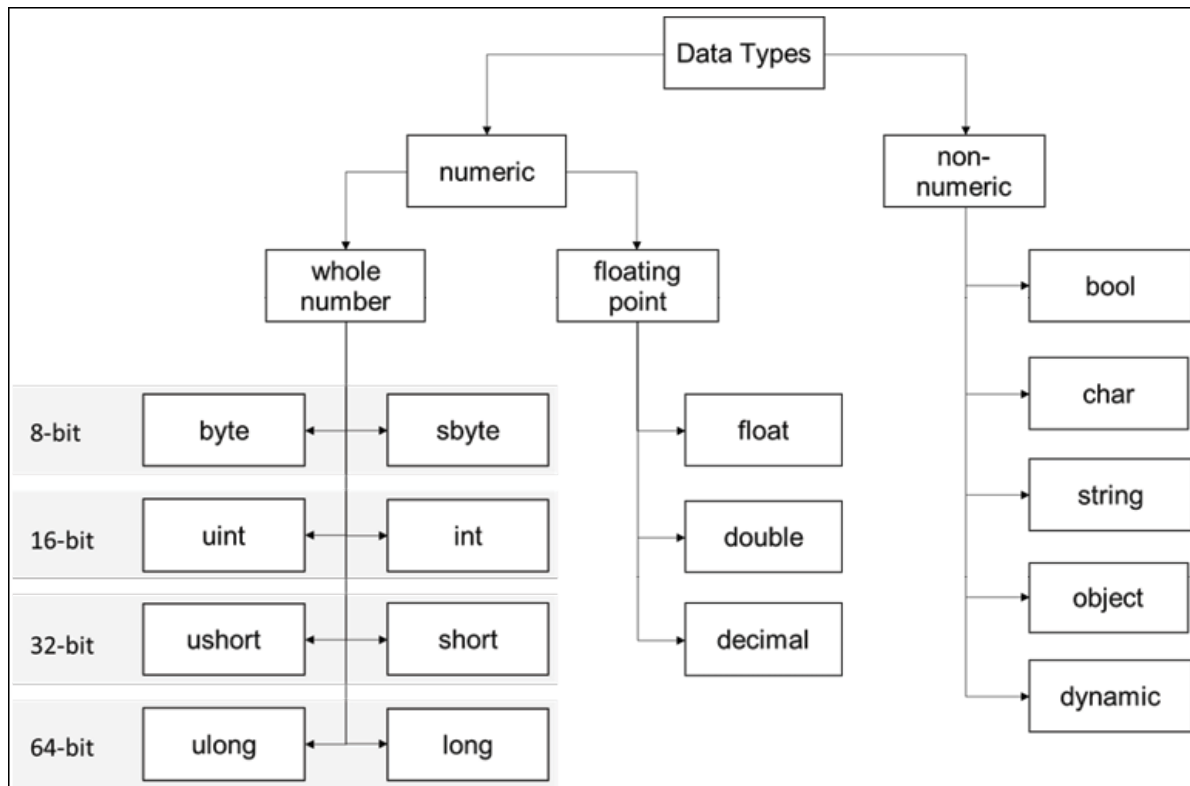


Figure 4.1: Data types in C#

As you can see, we have divided the data types primarily into two categories: numeric and non-numeric:

- In the non-numeric data types, we have **bool**, **char**, **string**, **object**, and **dynamic** types.
- In the numeric category, again, we have two branches: one for whole numbers and the other for fractional numbers.
 - Data types for fractional numbers may be **single**, **double**, or **decimal**.
 - For whole numbers, we have two sub-categories depending on whether we want both negative and positive numbers (signed) or only positive numbers (unsigned).

The following table indicates the .NET equivalent types, along with the memory taken up by each data type, along with the range of values that can be assigned:

C# Data Type	.NET equivalent	Description	Memory Size(bits)	Range	Default Value

sbyte	System.Sbyte	signed byte	8	-128 to 127	0
short	System.Int16	signed short integer	16	-32768 to 32767	0
Int	System.Int32	signed integer	32	-231 to 231-1	0
long	System.Int64	signed long integer	64	-263 to 263-1	0L
byte	System.Byte	unsigned byte	8	0 to 255	0
ushort	System.UInt16	unsigned short integer	16	0 to 65535	0
uint	System.UInt32	unsigned integer	32	0 to 232	0
ulong	System.UInt64	unsigned long integer	64	0 to 263	0

Table 4.1 : Overview of C# data types

Strings

In C#, a string is an object of **System.String** class, representing a sequence of Unicode characters. Strings are immutable, meaning their values cannot be changed after creation. Here is a simple example of creating and displaying a string in C#:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {

```

```

7.      // Creating a string
8.      string greeting = "Hello World!";
9.
10.     // Displaying the string
11.     Console.WriteLine(greeting);
12. }
13. }

```

Let us look at some common string operations that we come across in our programs:

- **String concatenation:** String concatenation is a common operation where two or more strings are combined to create a new string. This can be achieved using the **+** operator or the string **Concat** method. For example:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         // String concatenation using the + operator
8.         string firstName = "John";
9.         string lastName = "Doe";
10.        string fullName = firstName + " " + lastName;
11.
12.        // String concatenation using string.Concat method
13.        string greeting = string.Concat("Hello, ",
14.        , fullName, "!");

```

```

15.      // Displaying the concatenated strings
16.      Console.WriteLine(fullName);
17.      Console.WriteLine(greeting);
18.  }
19. }

```

- **String interpolation:** C# supports string interpolation, a concise and readable way to embed expressions and variables within a string. This is achieved using the **\$** symbol before the string and placing expressions or variables within curly braces **{}** as shown in the following example:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         // String interpolation
8.         string firstName = "John";
9.         string lastName = "Doe";
10.
11.        // Interpolated string
12.        string fullName = $"{firstName} {lastName}";
13.
14.        // Displaying the interpolated string
15.        Console.WriteLine(fullName);
16.    }
17. }

```

- **String methods and properties:** The **System.String** class in C# provides numerous methods and properties for string manipulation. A few commonly used ones are shown in the following examples:

```
1. string text = "Hello, World!";
2. int length = text.Length; // Returns
   the number of characters in the string.
3. string message = "MixedCase";
4. string upperCase = message.ToUpper(); // Convert
   the string to uppercase
5. string lowerCase = message.ToLower(); // Convert
   the string to lowercase
6. string original = "Hello, C#!";
7. string substring = original.Substring(7); // Extr
   acts a
   substring from the original string.
8. string phrase = "C# is amazing!";
9. int indexOfIs = phrase.IndexOf("is"); // Returns
   the index of
   the first occurrence of a specified character or
   substring.
```

- **String comparison:** When comparing strings, it is crucial to use appropriate methods to ensure accurate results. The **==** operator checks for equality of content, while the **String.Equals** method provides more options, including case-insensitive and culture-specific comparisons. Refer to the following code:

```
1. string str1 = "hello";
2. string str2 = "HELLO";
3.
4. bool areEqualUsingOperator = (str1 == str2); // f
   alse
5. bool areEqualUsingEquals = string.Equals(str1, st
```

```
r2, StringComparison.OrdinalIgnoreCase); // true
```

Operators and expressions

Operators are symbols that perform operations on variables and values in an expression. Operators in C# can be broadly classified into the following categories:

- **Assignment operators:** Denoted by = sign, this operator is used to assign a value or the result of an expression to a variable:

```
1. int num1 = 5;
```

```
2. int num2 = num1 * 5;
```

A self-assignment operator assigns a modified value to itself:

```
1. num1 += 10 // alternate way of writing num1 =  
   num1 + 10
```

When you want to increment or decrement the value by one, you can use prefix or postfix operators, too:

```
1. num1++;
```

```
2. ++num1;
```

- **Relational operators:** These are used to compare values, and the result of the comparison is true or false:

Operator	Description
==	Equal
!=	Not Equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Lesser than or equal to

Table 4.2 : Relational operators

- **Logical operators:** These can help evaluate multiple expressions and are normally used in combination with relational operators:

Operator	Description
&&	And
	Or
!	Not

Table 4.3 : Logical operators

- **Mathematical operators:** C# provides the following math operators for numerical expressions:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Table 4.4 : Mathematical operators

While using operators, it is important to take operator precedence into consideration as some operators will be evaluated first. When writing expressions involving multiple operators, it is always better to use parentheses () to group expressions to ensure proper results.

Branches and loops

Branching or conditional statements are used to determine whether a block of code gets executed. In C#, you can use any of the following:

- **If-else:** The if-else statement in C# provides a straightforward way to implement branching logic. Here is a simple example to illustrate the concept:

```
1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         int number = 10;
8.
9.         if (number > 0)
10.        {
11.            Console.WriteLine("The number is posi
tive.");
12.        }
13.        else
14.        {
15.            Console.WriteLine("The number is non-
positive.");
16.        }
17.    }
18. }
```

For scenarios involving multiple conditions, the **else-if** clause can be added to the **if-else** construct. This allows for a sequence of conditional checks, each with its associated code block. A better alternative is the Switch-Case block.

- **Switch-case:** The switch-case statement allows developers to compare the value of an expression against a list of possible values and execute the corresponding block of code. Let us consider a

scenario where a program needs to handle different days of the week and execute specific actions based on the day:

```
1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         // Day of the week (1: Monday, 2: Tuesday, ..
8.         ., 7: Sunday)
9.         int dayOfWeek = 3;
10.
11.         switch (dayOfWeek)
12.         {
13.             case 1:
14.                 Console.WriteLine("It's Monday!");
15.                 break;
16.             case 2:
17.                 Console.WriteLine("Happy Tuesday!");
18.                 break;
19.             case 3:
20.                 Console.WriteLine("Hello Wednesday!");
21.                 break;
22.             case 4:
23.                 Console.WriteLine("Almost there! It's
24. Thursday.");
25.                 break;
26.             case 5:
27.                 Console.WriteLine("TGIF! Friday!");
28.                 break;
29.             case 6:
30.             case 7:
31.                 Console.WriteLine("Happy Weekend!");
32.                 break;
33.             default:
```



```

32.             Console.WriteLine("Invalid day of the
           week.");
33.             break;
34.         }
35.     }
36. }

```

In this example:

- The **switch** statement evaluates the value of **dayOfWeek**.
- Cases 1 through 7 handle each day of the week, with specific messages printed for each day.
- The **default** case captures any value not covered by the defined cases.

Loops

Looping statements are used to repeat a block of code for a specified number of times or based on some condition. C# supports the following:

For Loop

The For loop is ideal for scenarios where the number of iterations is known in advance. Here are some key details regarding For loop:

The general syntax for a **for** loop in C# is as shown in the following code:

```

1. for (initialization; condition; iteration)
2. {
3.     // Code to be executed
4. }

```

The loop structure consists of three main components: initialization, condition, and iteration:

- **Initialization:** This part is executed only once at the beginning of the loop. It is typically used to initialize a loop control variable.
- **Condition:** The loop will continue executing as long as this condition is true. If the condition evaluates to false, the loop terminates.

- **Iteration:** This is executed after each iteration of the loop and is usually responsible for modifying the loop control variable.

Here is a simple example. Let us explore a scenario where we want to print the numbers from 1 to 5:

```

1. using System;
2. class Program
3. {
4.     static void Main()
5.     {
6.         // Using a for loop to print numbers from 1 to 5
7.         for (int i = 1; i <= 5; i++)
8.         {
9.             Console.WriteLine(i);
10.        }
11.    }
12. }

```

Let us break down this example:

- **Initialization:** `int i = 1;` initializes the loop control variable `i` to `1`.
- **Condition:** `i <= 5;` specifies that the loop should continue as long as `i` is less than or equal to `5`.
- **Iteration:** `i++` increments the value of `i` by `1` after each iteration.
- The output of this program will be the numbers 1 to 5.

Foreach

When dealing with collections such as arrays or lists, the **foreach** loop is a convenient option for iteration. It provides a simpler and more concise syntax compared to the 'for' loop without having to manage an index variable.

The following code provides a simple example of how the foreach loop is used in a program:

```

1. using System;
2.

```

```

3. class Program
4. {
5.     static void Main()
6.     {
7.         // Creating an array of strings
8.         string[] fruits = { "Apple", "Banana",
9.                             "Orange", "Grapes", "Mango" };
10.        // Using foreach to print each element in the
11.        array
12.        foreach (var fruit in fruits)
13.        {
14.            Console.WriteLine(fruit);
15.        }
16. }

```

Let us break down this example:

- The array of fruits is created with five string elements.
- The **foreach** loop iterates through each element in the fruits array, and during each iteration, the variable fruit takes on the value of the current element.
- The **Console.WriteLine(fruit)** statement prints each fruit to the console.

While and Do-while

The while and do-while loops provide flexible mechanisms for repetitive execution of code based on a given condition. These loops are particularly useful when the number of iterations is not known beforehand or when you want to ensure that a block of code is executed at least once.

The while loop is characterized by its simple and versatile structure. It repeats a block of code as long as a specified condition remains true. Let us look at a simple example:

```

1. using System;
2.

```

```

3. class Program
4. {
5.     static void Main()
6.     {
7.         // Using a while loop to print numbers from 1
           to 5
8.         int i = 1;
9.         while (i <= 5)
10.        {
11.            Console.WriteLine(i);
12.            i++;
13.        }
14.    }
15. }

```

In this example:

- The loop initializes a variable **i** to **1**.
- The condition **i <= 5** specifies that the loop should continue as long as **i** is less than or equal to **5**.
- The code inside the loop prints the value of **i** and increments it by **1** in each iteration.
- The output of this program will display numbers **1** to **5**.

The do-while loop is similar to the while loop but with a crucial difference: it always executes the code block at least once, even if the condition is initially false. Let us illustrate the do-while loop with an example. Here, we will take user input to guess a secret number:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         // Secret number to guess
8.         int secretNumber = 7;
9.         int guess;

```

```

10.
11.     // Ask the users for input until they guess c
    orrect number
12.     do
13.     {
14.         Console.Write("Guess the number: ");
15.         guess = int.Parse(Console.ReadLine());
16.
17.     } while (guess != secretNumber);
18.
19.     Console.WriteLine("You guessed the correct nu
    mber.");
20. }
21. }

```

Let us break down this example:

- The **do** block contains the code to repeatedly ask the user for input.
- The **while (guess != secretNumber)** condition ensures that the loop continues until the user guesses the correct number.
- Once the user guesses the correct number, the loop exits, and a congratulatory message is displayed.

Break and continue

The **break** statement is a loop control statement that allows you to prematurely exit a loop based on a certain condition. When executed, the break statement immediately terminates the loop, and control is transferred to the next statement following the loop.

Let us consider a scenario where we want to find the first occurrence of a particular number in an array using a for loop. Once the number is found, there is no need to continue the iteration. The **break** statement proves useful in such cases as the following example:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()

```

```

6.    {
7.        int[] numbers = { 5, 12, 8, 3, 10, 7 };
8.        int targetNumber = 8;
9.
10.       for (int i = 0; i < numbers.Length; i++)
11.       {
12.           if (numbers[i] == targetNumber)
13.           {
14.               Console.WriteLine($"Number {targetNumber} found at index {i}.");
15.               break; // Exit the loop once the number is found
16.           }
17.       }
18.   }
19. }

```

Let us break down this example:

- The for loop iterates through each element in the numbers array.
- The if statement checks if the current element is equal to the **targetNumber**.
- If a match is found, the loop is exited using the break statement.
- This ensures that the loop terminates when the target number is found, preventing unnecessary iterations.

The **continue** statement, on the other hand, allows you to skip the rest of the code within a loop for the current iteration and move on to the next one. This can be helpful when you want to bypass certain elements based on specific conditions. Consider a scenario where we want to print only the odd numbers from an array:

```

1. using System;
2.
3. class Program
4. {
5.     static void Main()
6.     {
7.         int[] numbers = { 5, 12, 8, 3, 10, 7 };

```

```

8.
9.     for (int i = 0; i < numbers.Length; i++)
10.    {
11.        if (numbers[i] % 2 == 0)
12.        {
13.            // Skip even numbers and continue to
            the next iteration
14.            continue;
15.        }
16.
17.        Console.WriteLine($"Odd number found: {nu
18.        mbers[i]}");
19.    }
20. }

```

Let us break down this example:

- The **for** loop iterates through each element in the **numbers** array.
- The **if** statement checks if the current element is an even number using the condition **numbers[i] % 2 == 0**.
- If an even number is encountered, the **continue** statement is executed, skipping the **Console.WriteLine** statement and move to the next iteration.
- This results in the program printing only the odd numbers from the array.

Classes, methods, and properties

C# is a versatile programming language that supports **object-oriented programming (OOP)** principles. Classes, methods, and properties are fundamental building blocks of object-oriented design, enabling developers to create modular, reusable, and well-organized code. Let us explore these concepts with a simple example.

Classes

In C#, a class is a blueprint for creating objects. Objects are instances of classes, and they encapsulate data and behavior. The structure of a class includes fields, properties, methods, and other members. Here is a basic example of a **Person** class:

```
1. using System;
2.
3. class Person
4. {
5.     // Fields
6.     public string FirstName;
7.     public string LastName;
8.
9.     // Method to display full name
10.    public void DisplayFullName()
11.    {
12.        Console.WriteLine($"Full Name: {FirstName} {L
astName}");
13.    }
14. }
```

Let us break down this example:

- The **Person** class has two fields (**FirstName** and **LastName**) to store individual's names.
- It contains a method (**DisplayFullName**) that prints the person's full name to the console.

Methods

Methods in C# encapsulate actions and behaviors that objects can perform. They define the functionality associated with a class. Let us create an instance of the **Person** class and invoke the **DisplayFullName** method:

```
1. class Program
2. {
3.     static void Main()
4.     {
5.         // Creating an instance of the Person class
```



```

6.         Person person = new Person();
7.
8.         // Setting values for the fields
9.         person.FirstName = "John";
10.        person.LastName = "Doe";
11.
12.        // Invoking the DisplayFullName method
13.        person.DisplayFullName();
14.    }
15. }

```

Let us break down this example:

- An instance of the **Person** class named **person** is created.
- Values are assigned to the **FirstName** and **LastName** fields of the **person** object.
- The **DisplayFullName** method is invoked on the **person** object, resulting in the output: **Full Name: John Doe**.

Properties

Properties provide a way to encapsulate and control access to the data within a class. They allow for the definition of get and set accessors, enabling fine-grained control over data manipulation. Let us enhance our **Person** class with properties:

```

1. class Person
2. {
3.     // Properties
4.     public string FirstName { get; set; }
5.     public string LastName { get; set; }
6.
7.     // Method to display full name
8.     public void DisplayFullName()
9.     {
10.        Console.WriteLine($"Full Name: {FirstName} {L
    astName}");
11.    }
12. }

```

Now, the **FirstName** and **LastName** fields have been replaced with properties, providing a more controlled way to access and modify the data.

Inheritance and interface

Inheritance is a cornerstone of object-oriented principles, fostering code reuse, modularity, and a hierarchical organization of classes. Inheritance empowers developers to create new classes based on existing ones, facilitating the construction of efficient and scalable code.

At its core, inheritance enables a new class (called the derived class or child class) to inherit properties and behaviors from an existing class (called the base class or parent class). This allows developers to model relationships between entities and promote code reuse, reducing redundancy and enhancing maintainability. Here is a simple example to illustrate the concept:

```
1. class Animal
2. {
3.     public string Name { get; set; }
4.
5.     public void Eat()
6.     {
7.         Console.WriteLine($"{Name} is eating.»");
8.     }
9. }
10.
11. class Dog : Animal
12. {
13.     public void Bark()
14.     {
15.         Console.WriteLine($"{Name} is barking.»");
16.     }
17. }
```

Let us break down this example:

- The **Animal** class serves as the base class, defining a property **Name** and a method **Eat**.

- The **Dog** class is a derived class that inherits from **Animal**. It introduces a new method **Bark**.

Inheritance allows developers to extend the functionality of a base class in a derived class. In our example, a **Dog** can perform actions specific to dogs, such as barking, while inheriting the general attributes and behaviors from the **Animal** class:

```

1. class Program
2. {
3.     static void Main()
4.     {
5.         // Creating an instance of the Dog class
6.         Dog myDog = new Dog { Name = "Buddy" };
7.
8.         // Using inherited methods from the Animal class
9.         myDog.Eat();
10.
11.        // Using methods specific to the Dog class
12.        myDog.Bark();
13.    }
14. }

```

Let us break down this example:

- An instance of the **Dog** class is created, and the **Name** property is set to "**Buddy**".
- The **Eat** method, inherited from the **Animal** class, is called on the **Dog** object.
- The **Bark** method, specific to the **Dog** class, is also called.

Interfaces

In C#, interfaces play a pivotal role in achieving abstraction and encapsulation providing a blueprint for achieving multiple inheritance. Interfaces enable developers to define a set of methods and properties that classes must implement, promoting a consistent and contract-based approach to programming. An interface is a contract that defines a set of methods and properties without specifying their implementation. It serves

as a way to establish a common ground for different classes to adhere to a shared set of behaviors. The syntax for declaring an interface is as follows:

```
1. interface IShape
2. {
3.     void Draw();
4.     double CalculateArea();
5. }
```

Let us break down this example:

- **IShape** is an interface with two methods: **Draw** and **CalculateArea**.
- The interface does not provide any implementation details; it only defines the signatures of the methods.

Classes that implement an interface must provide concrete implementations for all the methods declared in that interface. Let us create a **Circle** class that implements the **IShape** interface:

```
1. using System;
2.
3. class Circle : IShape
4. {
5.     public double Radius { get; set; }
6.
7.     // Implementation of Draw method
8.     public void Draw()
9.     {
10.         Console.WriteLine("Drawing a circle");
11.     }
12.
13.     // Implementation of CalculateArea method
14.     public double CalculateArea()
15.     {
16.         return Math.PI * Radius * Radius;
17.     }
18. }
```

Let us break down this example:

- The **Circle** class implements the **IShape** interface by providing concrete implementations for the **Draw** and **CalculateArea** methods.
- The **Radius** property is specific to the **Circle** class and is not part of the interface.

Utilizing interface polymorphism: One of the powerful aspects of interfaces is that they enable polymorphism. This means that you can treat objects that implement the same interface interchangeably. Let us demonstrate this by creating a method that takes an **IShape** parameter:

```

1. class Program
2. {
3.     static void DisplayShapeInfo(IShape shape)
4.     {
5.         shape.Draw();
6.         Console.WriteLine($"Area: {shape.CalculateArea()}");
7.     }
8.
9.     static void Main()
10.    {
11.        // Creating a Circle object
12.        Circle circle = new Circle { Radius = 5.0 };
13.
14.        // Displaying information using the DisplayShapeInfo method
15.        DisplayShapeInfo(circle);
16.    }

```

Let us break down this example:

- The **DisplayShapeInfo** method takes an **IShape** parameter, allowing it to accept any object that implements the **IShape** interface.
- We create a **Circle** object and pass it to the **DisplayShapeInfo** method, demonstrating that the method can work with any object adhering to the **IShape** contract.

Delegates and events

In C#, delegates are powerful constructs that enable the implementation of function pointers, callback mechanisms, and event handling. They provide a flexible and type-safe way to encapsulate method references, allowing developers to achieve decoupling and promote modular design.

In essence, a delegate is a type that represents references to methods. It allows methods to be passed as parameters, stored as variables and invoked dynamically. Delegates are especially useful when dealing with events, asynchronous programming, and creating extensible frameworks.

Let us explore a straightforward example using a delegate to define a method signature for a simple arithmetic operation:

```
1. using System;
2.
3. // Declare a delegate for an arithmetic operation
4. delegate int ArithmeticOperation(int x, int y);
5.
6. class Calculator
7. {
8.     // Method to add two numbers
9.     static int Add(int x, int y)
10.    {
11.        return x + y;
12.    }
13.
14.    // Method to subtract two numbers
15.    static int Subtract(int x, int y)
16.    {
17.        return x - y;
18.    }
19. }
20.
21. class Program
22. {
23.     static void Main()
24.     {
```

```

25.         // Create instances of the delegate, pointing
           to
           the Add and Subtract methods
26.         ArithmeticOperation addDelegate = Calculator.
           Add;
27.         ArithmeticOperation subtractDelegate = Calcul
           ator.Subtract;
28.
29.         // Perform arithmetic operations using the de
           legates
30.         int resultAdd = addDelegate(5, 3);
31.         int resultSubtract = subtractDelegate(8, 4);
32.
33.         Console.WriteLine($"Addition Result: {resultA
           dd}");
34.         Console.WriteLine($"Subtraction Result: {resu
           ltSubtract}");
35.     }
36. }

```

Let us break down this example:

- The **ArithmeticOperation** delegate is declared, specifying a method signature that takes two integers and returns an integer.
- The **Calculator** class contains static methods (**Add** and **Subtract**) matching the delegate's signature.
- Instances of the delegate are created, pointing to the **Add** and **Subtract** methods.
- The delegate instances are then invoked with specific parameters, demonstrating how delegates act as function pointers.

Events

Events are a crucial component for facilitating communication between different parts of a program. Events allow one class (known as the publisher) to notify other classes (known as subscribers) when a specific action or state change occurs.

In .NET MAUI, building user interfaces involves extensive use of events. For example, a Button's click event is useful for capturing user interaction and triggering specific actions in response. Here is a simple example of a .NET MAUI program featuring a button that responds to a click event:

```
1. using Microsoft.Maui.Controls;
2.
3. public partial class MainPage : ContentPage
4. {
5.     public MainPage()
6.     {
7.         InitializeComponent();
8.
9.         // Create a button
10.        var clickMeButton = new Button
11.        {
12.            Text = "Click Me!",
13.            HorizontalOptions = LayoutOptions.Center,
14.            VerticalOptions = LayoutOptions.CenterAnd
Expand
15.        };
16.
17.        // Subscribe to the button's Clicked event
18.        clickMeButton.Clicked += OnClickMeButtonClick
ed;
19.
20.        // Add the button to the page
21.        Content = new StackLayout
22.        {
23.            Children = { clickMeButton }
24.        };
25.    }
26.
27.    // Event handler for the button's Clicked event
28.    private void OnClickMeButtonClicked(object sender
, EventArgs e)
29.    {
```



```

30.         // Respond to the button click
31.         DisplayAlert("Button Clicked", "You clicked t
    he button!", "OK");
32.     }
33. }

```

In this .NET MAUI example:

- A **Button** named **clickMeButton** is created and customized with a label and layout options.
- The **OnClickMeButtonClicked** method is designated as the event handler for the button's **Clicked** event.
- The **StackLayout** is employed to organize the button within the page.
- When the button is clicked, a simple alert is displayed, signaling the event response.

Arrays and lists

Arrays and lists serve as indispensable collections for efficiently storing and managing collections of data. They both provide distinct advantages and cater to various programming scenarios. Let us look at them in detail:

- **Arrays:** Arrays in C# represent fixed-size collections of elements of the same type. Declaring an array involves specifying the data type of its elements and the size of the array. Let us look at some examples to see how arrays are commonly used:

```

1. int[] numbers = new int[5]; // declaring an
    integer array with a size of 5
2. // Initializing at declaration
3. int[] numbers = { 1, 2, 3, 4, 5 };
4. // Initializing later
5. int[] numbers = new int[] { 1, 2, 3, 4, 5 };

```

- **Lists:** Lists provide dynamic collections that can grow or shrink in size at runtime. Lists are part of the **System.Collections.Generic** namespace and offer a more flexible alternative to arrays.

```
1. List<int> numbersList = new List<int>();  
   // declaring a list of integers  
  
2. List<int> numbersList = new List<int> { 1, 2, 3,  
   4, 5 };  
   // initializing a list of integers
```

The key differences between an array and a list are:

- Arrays have a fixed size determined at the time of declaration, whereas lists provide dynamic sizing, enabling elements to be added or removed without the need for predefining the size.
- Arrays generally have better performance for direct access to elements due to their contiguous memory allocation. Lists may incur a slight performance overhead due to their dynamic resizing and memory management.
- Arrays provide fewer built-in methods and features compared to lists. Lists offer a rich set of methods, such as **Add**, **Remove**, **Insert**, and more, making them convenient for various operations on collections.

Use arrays when:

- The size of the collection is known and remains constant.
- Direct, high-performance access to elements is crucial.
- Memory efficiency is a priority.

Use lists when:

- The size of the collection can change dynamically.
- Frequent additions or removals of elements are expected.
- Convenient methods for collection manipulation are required.

Conclusion

In conclusion, this crash course in C# has served as a rapid yet robust introduction to fundamental programming concepts in C#. From types, variables, and strings to branching, looping, and advanced topics like classes, methods, delegates, and events, you have prepared a solid foundation for your journey into C# programming. It is essential to note

that while this chapter provided a concise overview, the vast landscape of C# holds more in-depth knowledge waiting to be explored. Consider this crash course as the initial stepping stone in your journey, preparing you for deeper dives into the language's intricacies. With this newfound understanding, you should be well-equipped to venture further into the world of C# and unlock its full potential.

In the next chapter, we will explore XAML, a declarative language that facilitates the development of user interfaces and empowers the creation of concise code for various functionalities.

Points to remember

Here are some key take aways from this chapter:

- Understanding and using the right data type is critical when declaring variables.
- Strings are fundamental for handling textual data, and string interpolation and formatting enhance the readability and flexibility of data presented to users.
- While using operators in expressions, take operator precedence into consideration to avoid bugs.
- Proper use of branching constructs like if-else and switch-case enables logical decision-making in code.
- While choosing the right loop construct for repetitive tasks, ensure to have the exit condition as well to avoid infinite loops.
- Classes encapsulate data and behavior and help contribute to code modularity and organization.
- Implementing interfaces allows for standardized behavior across diverse classes.
- Delegates serve as powerful callback mechanisms allowing for event-driven programming.
- Arrays offer fixed-size collections, whereas lists provide dynamic sizing.

CHAPTER 5

Introduction to XAML

Introduction

In a .NET MAUI app, the **user interface (UI)** is basically constructed as a tree of objects. The UI is generally created using a markup language instead of C#. This markup language is an XML-based language and is called **eXtensible Application Markup Language (XAML)**. The markup defines the .NET MAUI classes and property values. At runtime, the XAML is parsed, and the objects are instantiated and initialized.

Structure

In this chapter, we will discuss the following topics:

- Overview of XAML
- XAML syntax
- Namespaces
- Markup extensions
- Passing arguments
- Data binding

Objectives

By the end of this chapter, you will gain a clear understanding of:

- Basic concepts of XAML
- The syntax of XAML
- Different types of constructs like object-element syntax, property-element syntax, attached property syntax, and markup extensions.

Overview of XAML

XAML was originally developed for Silverlight by *Rob Relyea* and the team at Microsoft. As it is a very powerful and productive way for developers to build rich user interfaces, it has been adopted in many other technologies, including WP7, WP8, UWP, WinUI, Xamarin, and also in .NET MAUI. Though apps can be developed completely using C#, defining the UI in XAML is recommended because of several advantages.

The major advantages that XAML provides are the following:

- It allows you to separate the UI design from the code behavior.
- XAML is more compact when compared to implementing the same functionality in C#.
- XAML provides a visual representation of the UI structure.
- XAML has a great tooling support.
- XAML makes it possible to use powerful concepts such as data binding.

Each Page/View in a .NET MAUI file can be defined in XAML, and a companion **.cs** file is used to define the event handlers and methods containing the programming logic.

XAML syntax

The syntax of XAML is similar to XML, with some additional features that follow these basic rules as in XML:

- The first line starts with an XML prolog, which is provided by default in all XAML files:
`<?xml version="1.0" encoding="utf-8" ?>`

- The rest of the XAML file consists of elements organized in a tree structure.

Refer to the following figure:

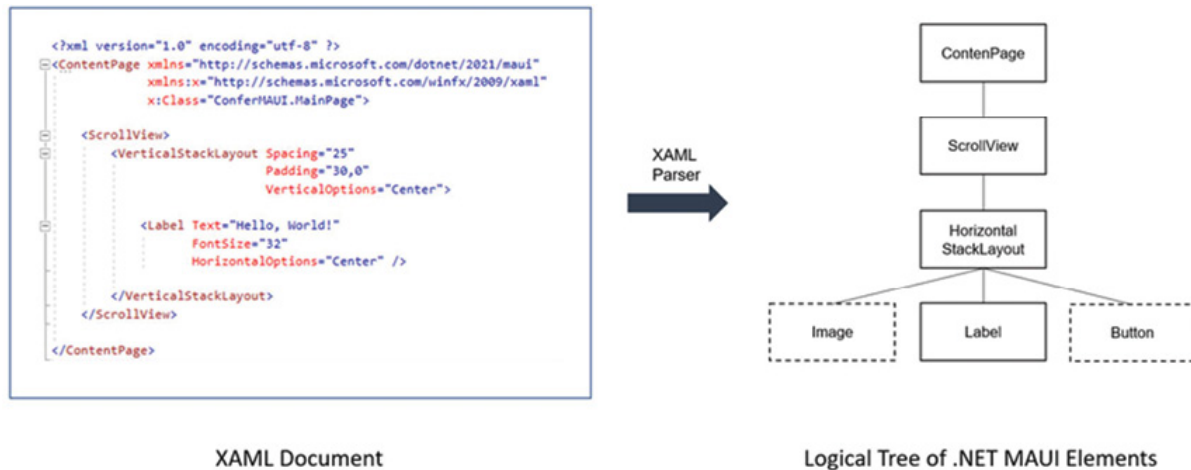


Figure 5.1: Logical Tree created from the XAML markup

- To create a user interface, the elements must be nested properly. The way the elements are nested determines the structure of the UI.
- The outermost element is called the root element and contains all the other elements.
- Each XAML file must contain exactly one root element, which is the parent of all other elements. In .NET MAUI, except for the **App.xaml** and **AppShell.xaml** files, the root element in all other XAML Files will be either a **ContentPage**, **ContentView** or **ResourceDictionary**.
- XAML elements represent objects in the UI. For example, a simple UI with four objects has four XAML elements that correspond to each of the objects in the logical tree.

Refer to the following figure:

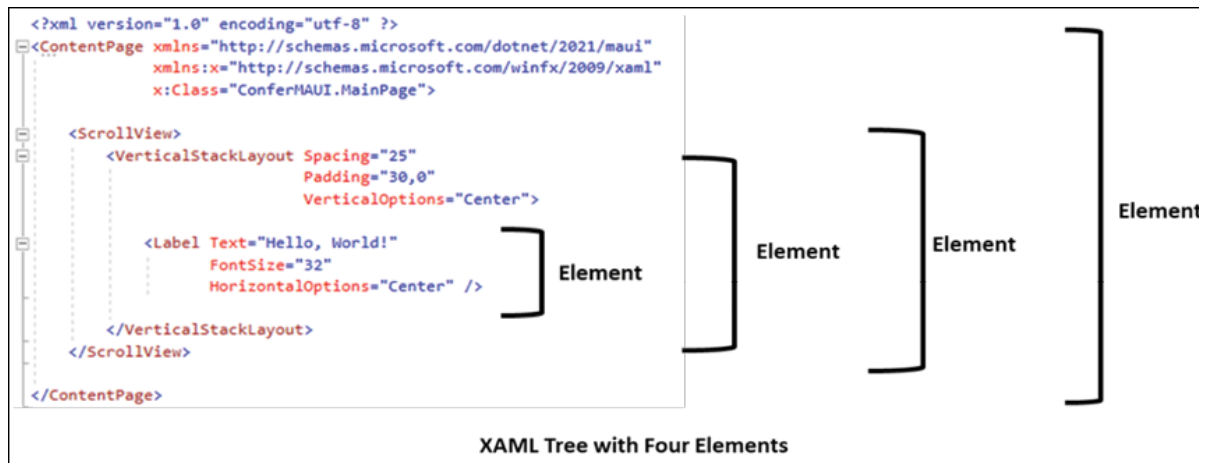


Figure 5.2: Nesting of elements in XAML

Object element syntax

Using XAML elements to represent .NET MAUI class objects is called object element syntax because each XAML element represents a .NET MAUI object.

The default syntax for an element has three parts: the start tag, the content section, and the end tag, as shown in the following figure:

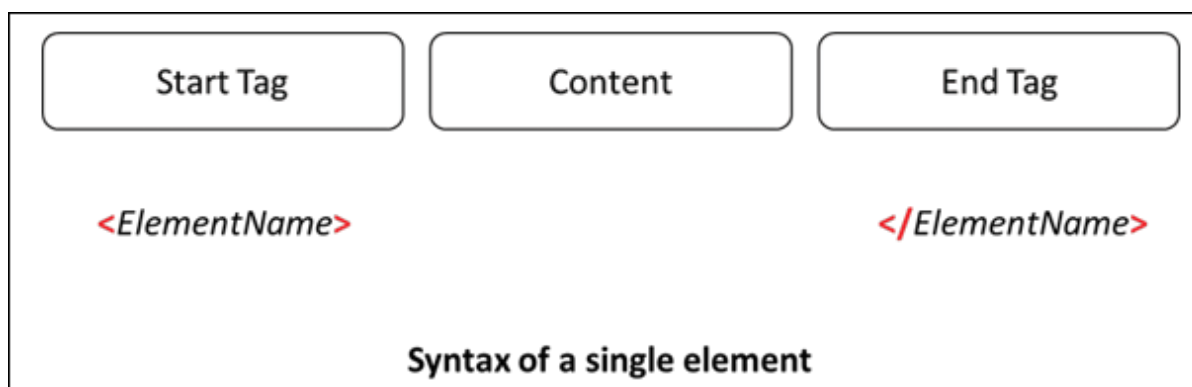


Figure 5.3: Syntax of a single element

- The opening tag consists of the element name wrapped in angle brackets e.g. **<StackLayout>**
- The end tag will have a slash just before the element name e.g. **</StackLayout>**
- The content section can have text, other elements or white space

Comments

For commenting, use the following syntax:

1. `<!-- This is a comment -->`

Attribute syntax

Remember each element corresponds to a class object. You can set the properties of the class object using attributes. For example:

1. `<Button Width="75" Height="25" />`

In this example, it sets the values of two properties of a **Button** object: **Width** and **Height**:

- Attributes must be placed inside the start tag, after the element name, and cannot appear in the content section or end tag.
- The syntax of an attribute contains an identified followed by an = (equal to) sign.
- The property values must be set using double quotes.
- An element may have any number of attributes. If there are multiple attributes, they must be separated by a white space.

Refer to the following figure:

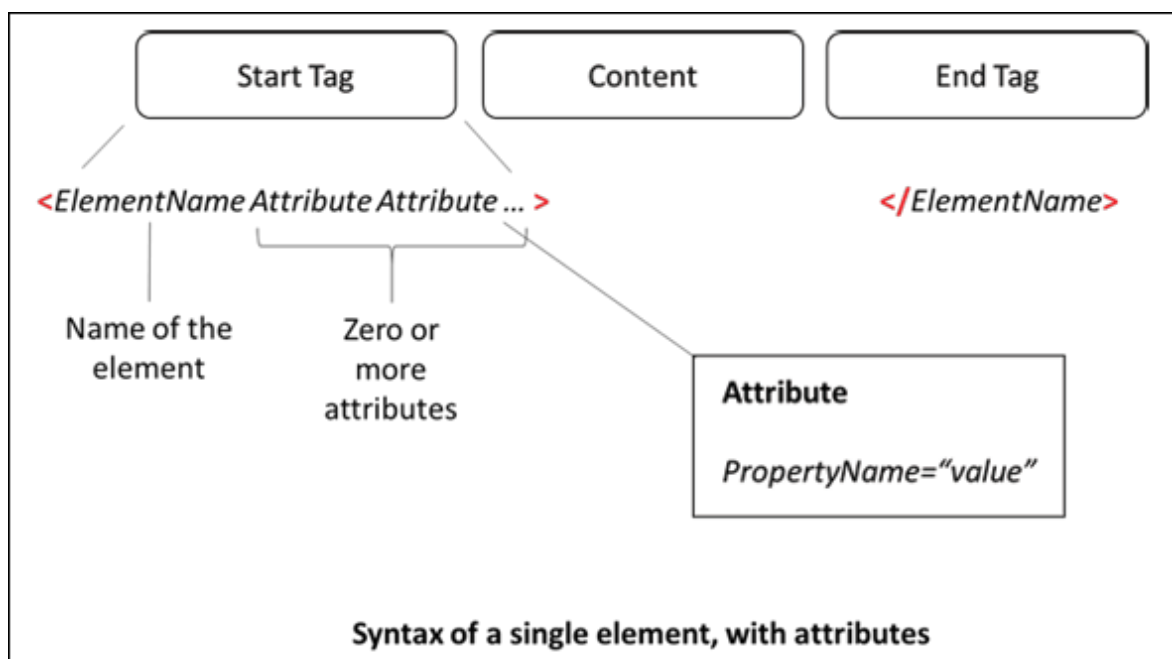


Figure 5.4: Syntax of a single element, with attributes

Empty element syntax

In case the element does not have any content, you can specify it in two ways:

- You can leave the content section blank or provide whitespace between the start and end tags:

1. `<Button Width="100" Height="50"></Button>`

Or,

- Use the empty element syntax, in which you have only a single tag, instead of the start and end tag. It resembles the start tag but, at the end of the tag, it uses `/|` instead of `>|` as the delimiter.

For example, refer to this command:

2. `<Button Width="100" Height="50" />`

Except the XML prolog, all elements must have either an end tag or a self-closing tag using the empty element syntax.

Property value syntax

For simple property values, attributes are helpful. For setting complex property values, we can use the property element syntax in the following way:

The property is not mentioned as an attribute but nested in the content section of the element.

For example, refer to this command:

1. `<ElementName.PropertyName> Value </ElementName.PropertyName>`

- This uses dot-syntax notation; the first part is the name of the class, and the second part is the name of the property.
- The XAML parser does not instantiate an object in this case, but sets the property on the object.
- The property element tag cannot contain attributes.

The following example shows how this is useful.

- Attribute syntax:

1. `<VerticalStackLayout Background="LightBlue">`

2. `</VerticalStackLayout>`

Property-element syntax:

```
1. <VerticalStackLayout>
2.     <VerticalStackLayout.Background>
3.         LightBlue
4.     </VerticalStackLayout.Background>
5. </VerticalStackLayout>
```

In the above example, when the Background color property value is simple, attribute syntax is easier. However, if the property value is complex, like the following example, then property element syntax will have to be used:

In .NET MAUI we can apply a gradient color to the **Background** property using a **LinearGradientBrush**. This requires you to set two **GradientStops** with different colors, and .NET MAUI will take care of providing a smooth gradient between the two colors. Refer to the following code:

```
1. <VerticalStackLayout>
2.     <VerticalStackLayout.Background>
3.         <LinearGradientBrush StartPoint="0,0"
4.             EndPoint="1,1">
5.             <GradientStopCollection>
6.                 <GradientStop Offset="0"
7.                     Color="Red" />
8.                 <GradientStop Offset="1"
9.                     Color="Yellow" />
10.            </GradientStopCollection>
11.        </LinearGradientBrush>
12.    </VerticalStackLayout.Background>
13. </VerticalStackLayout>
```

Attached property syntax

Attached properties are special in that it is defined in one class but used in another. Let us look at an example:

```
1. <Grid>
2.     <Grid.RowDefinitions>
3.         <RowDefinition Height="*" />
4.         <RowDefinition Height="*" />
```

```
5.     </Grid.RowDefinitions>
6.     <Button Grid.Row="1" />
7. </Grid>
```

In this above example, the **Button** has an attribute assignment for **Grid.Row** property. This is not part of the **Button** class, but the **Row** property is defined in the **Grid** class. This is what is called an attached property.

When in doubt, observe the class name. If it is different from the class being instantiated, it is an attached property. In property element syntax, the class name is the same as the object being instantiated.

Namespaces

As mentioned earlier in the chapter, each part of an XAML tree corresponds to a specific .NET MAUI class. When the XAML parser needs to create objects of these types, it must know where to find their definitions. This information is communicated to the parser through something called XAML namespaces.

A XAML namespace is like a special name that represents a group of .NET MAUI classes. If you want to use a class as an element in your XAML markup, you need to tell the XAML parser which namespace the class belongs to.

Using namespaces in XAML is not like using namespaces in C#. In C#, you include namespaces with a bunch of using statements at the top of the code file, and then you can use the classes from those namespaces in your code (as long as you have a reference to the assembly containing the namespace).

In XAML, it is a bit different. Only one namespace can be the default. You can set the default namespace using the **xmlns** keyword followed by an equal sign and then a string that is the name of the default namespace. The strings look like website addresses, but they are not.

Other namespaces can be declared but need to be provided with a prefix. For this, you use the keyword **xmlns:**, followed by a colon and a character or string, followed by an equal sign and the name of the namespace. The prefix that you declare will have to be mentioned wherever the classes or types from the respective namespace are being used.

In the following code, you can see the namespaces defined in the beginning of every XAML file:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
3.           xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
4.           x:Class="Some Class Name"
5.           Title="Some Title">
6.
7. </ContentPage>
```

Besides these, you may also add other namespaces, or from your own project. If you want to use a namespace that is not a XAML namespace, there is a special syntax to tell the XAML parser to use a **common language runtime (CLR)** namespace like this:

```
1.           xmlns:local="clr-namespace:MyNameSpace"
```

Alternatively, you can write it as the following:

```
1.           xmlns:local="using:MyNameSpace"
```

In case the namespace is in another assembly, then you can include the assembly name as well:

```
1.           xmlns:local="clr-
   namespace:MyNameSpace; assembly=MyAssembly"
```

Object names

As mentioned earlier in the chapter, XAML is used to instantiate the objects in the UI of a .NET MAUI app. The actions performed by the objects are defined in a companion C# file, known as the code-behind file.

By default, XAML instantiates objects without a name, for example:

```
1. <Button Text="Click"></Button>
```

However, in C#, whenever objects are created, a name has to be provided, for example:

```
1. Button btn = new Button();
```

The name is required to define further actions and behavior in the code-behind file.

When creating objects in XAML, you can provide a name by using the **x:Name** attribute:

```
1. <Button x:Name="btn" Text="Click" />
```

The **x** is the prefix mentioned in the standard namespace of the XAML document, which is used for elements and attributes that are intrinsic to XAML.

Markup extensions

While XAML is very powerful, there are some limitations. Let us understand these limitations and how we can deal with them:

- The first limitation is that XAML cannot evaluate values and perform conditional logic during runtime. For instance, let us imagine we have a Label in our UI, and we want to display AM if the time is before noon and PM if it is post-noon. In XAML, there is no direct way to perform this conditional logic at runtime to set the text accordingly.
- Another limitation is that when assigning a value to a property in XAML, we can use a value type or create a new object. XAML lacks a syntax for assigning an existing, non-static object to a property.

These are just a couple of limitations where pure XAML just falls short. To handle such situations and others requiring runtime evaluation, XAML needs a *hook* to an external code that is callable at runtime. The code must execute the necessary actions and return the value or object reference needed for XAML to complete its property assignment. This functionality is provided by something called a markup extension.

A markup extension acts as a connection to a class external to XAML, known as an extension class. The extension class is designed to be used by the markup extension.

In XAML, a markup extension is written by replacing the string on the right side of the equals sign in the attribute syntax. It begins and ends with curly braces.

Here is a basic form of a markup extension:

```
1. <Button Style="{StaticResource MyStyle}" />
```

In the above example, we are setting the `Style` property of the `Button` using a markup extension called **StaticResource**. It searches the program's resources for a style named **MyStyle** and, if found, returns its reference, which is then assigned to the **Style** property.

Markup extensions follow specific syntax forms, as shown in *Figure.5.5*. The name of the extension class is the first string inside the open curly brace, and there is no comma following the class name. After the class name, there can be zero or more strings separated by commas. These strings can be either simple strings or property/value pairs.

Refer to the following figure:

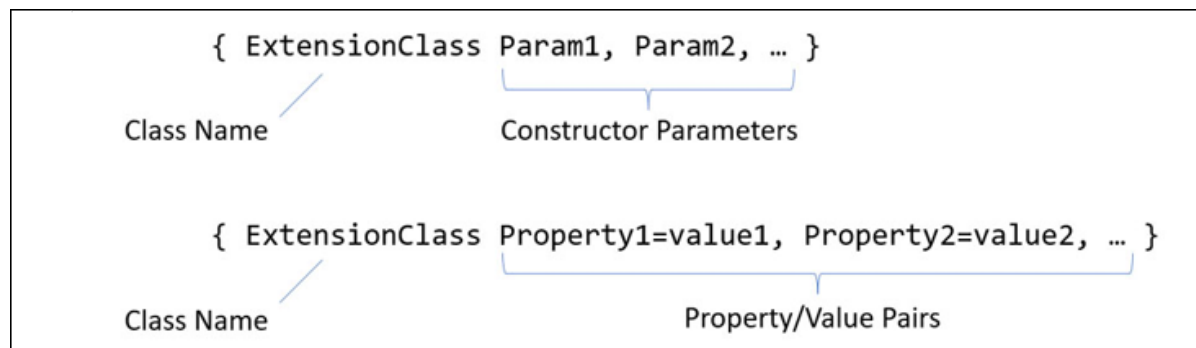


Figure 5.5: Two syntax forms for markup extensions

Passing arguments

In .NET MAUI, sometimes, when creating objects in XAML, you might need to use constructors that need specific information. For this, you can make use of **x:Arguments**.

x:Arguments is used to tell the program what information to give to the constructor when creating the object. For example, when creating a color object, you can use the **x:Arguments** to provide details like how much red, green, and blue it should have. Your number should match the color needs, like grayscale or different color channels. The following XAML markup represents a **Grid** element with a background color defined by a **Color** object that takes three specified values (0.25, 0.4, 0.35) as arguments that define the RGB color components.

1. `<Grid>`
2. `<Grid.Background>`
3. `<Color>`
4. `<x:Arguments>`

```

5.         <x:Color>0.25</x:Color>
6.         <x:Color>0.4</x:Color>
7.         <x:Color>0.35</x:Color>
8.     </x:Arguments>
9. </Color>
10. </Grid.Background>
11.</Grid>

```

Additionally, if you are working with generic types (i.e., types that can work with different kinds of data), you can use `x:TypeArguments` to give specific details. Arguments can be passed to constructors using the following .NET MAUI XAML language primitives:

- **x:Array**, for **Array**.
- **x:Boolean**, for **Boolean**.
- **x:Byte**, for **Byte**.
- **x:Char**, for **Char**.
- **x:DateTime**, for **DateTime**.
- **x:Decimal**, for **Decimal**.
- **x:Double**, for **Double**.
- **x:Int16**, for **Int16**.
- **x:Int32**, for **Int32**.
- **x:Int64**, for **Int64**.
- **x:Object**, for the **Object**.
- **x:Single**, for **Single**.
- **x:String**, for **String**.
- **x:TimeSpan**, for **TimeSpan**.
- **x:FactoryMethod** is used to specify a special method that can be used to set up an object. This special method must be a public static method that returns objects or values of the same type as the class that defines the method.

Let us look at an example:

```

1. <Grid>
2.     <Grid.Background>
3.         <Color x:FactoryMethod="FromRgba">
4.             <x:Arguments>
5.                 <x:Byte>192</x:Byte>
6.                 <x:Byte>75</x:Byte>
7.                 <x:Byte>150</x:Byte>
8.                 <x:Byte>128</x:Byte>
9.             </x:Arguments>
10.        </Color>
11.    </Grid.Background>
12. </Grid>

```

Data binding

In most .NET MAUI apps, the UI represents some data and allows users to modify it. Moreover, when the underlying data changes, we would like the UI to be updated as well. In .NET MAUI, this is made possible easily using data binding.

Data binding, as the name suggests, is a way to connect the data in your application with the user interface elements. It ensures that the changes in data automatically reflect in the UI and vice versa without writing a lot of extra code. It makes the app more responsive and dynamic.

Let us first discuss a situation where you want to bind a **TextBox** with a Slider control. You would like the **TextBox** to show the value of the Slider, and when you move the slider, the **TextBox** should update to match.

One way to do this is to write code in the code-behind file to handle events when the Slider changes. Instead, data binding provides a much simpler approach and does the heavy lifting for you.

For data binding to work, we need a **Source** property and a **Target** property, and then we can implement a data binding between the two:


```

1. public Demo1()
2. {
3.     InitializeComponent();
4.     label.BindingContext = slider;
5.     label.SetBinding(Label.ScaleProperty, "Value");
6.         // Alternatively, you can write a single stat
           ement like this:
7.     label.SetBinding(Label.ScaleProperty, new Binding("V
           alue", source: slider));
8. }

```

It is even simpler to implement the data binding in XAML like the markup shown in the following code:

```

1.     <StackLayout Padding="10, 0">
2.         <Label x:Name="label"
3.             Text="TEXT"
4.             FontSize="48"
5.             HorizontalOptions="Center"
6.             VerticalOptions="Center">
7.             <Label.Scale>
8.                 <Binding Source="
           {x:Reference slider}"
9.                     Path="Value" />
10.            </Label.Scale>
11.        </Label>
12.
13.        <Slider x:Name="slider"
14.            Maximum="360"
15.            VerticalOptions="Center" />
16.    </StackLayout>

```

Data binding is not just useful for binding UI elements to each other. It can also be used to bind the UI element to the model. To illustrate this, let us consider a **Person** class with two properties:

```

1. public class Person
2. {
3.     public string Name { get; set; }

```

```

4.
5.     public string Email { get; set; }
6. }

```

If we want to display the details of an object of the **Person** class in the UI, let us do it by adding two **Entry** elements to display the values:

```

1. <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
2.           xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
3.           x:Class="DataBindingSessionDemos.Demo"
4.           Title="Demo">
5.     <VerticalStackLayout Spacing="20">
6.         <Entry x:Name="lblName"
7.               VerticalOptions="Center"
8.               HorizontalOptions="Center" />
9.         <Entry x:Name="lblEmail"
10.              VerticalOptions="Center"
11.              HorizontalOptions="Center" />
12.     </VerticalStackLayout>
13. </ContentPage>

```

First, we will see how the data binding is done in C# in the code-behind file. Follow these steps:

1. For each UI element that we want to bind to data, we need a **Binding** object to be created by assigning the **Path** and the **Source** properties.
2. Then, we can invoke the **SetBinding** method of the elements and pass the required arguments—first, the property of the target to which we want to bind and second, the binding object that we just created in the previous step.

Refer to the following code:

```

1. public partial class Demo : ContentPage
2. {
3.     Person p1;
4.     public Demo()

```

```

5.     {
6.         InitializeComponent();
7.         p1 = new Person { Name = "John Doe", Email =
            "j.doe@msn.com" };
8.
9.         var nameBinding = new Binding
10.        {
11.            Path = "Name",
12.            Source = p1
13.        };
14.        lblName.SetBinding(Entry.TextProperty, nameBi
            nding);
15.
16.        var emailBinding = new Binding
17.        {
18.            Path = "Email",
19.            Source = p1
20.        };
21.
22.        lblEmail.SetBinding(Entry.TextProperty, email
            Binding);
23.    }
24.
25. }

```

Now, let us do the data binding in XAML. You will see how XAML syntax really shines through and makes the code simpler.

In the UI markup, only one change is required. In the two **Entry** elements, use the data binding syntax to map the two properties of the object to the **Text** properties of the **Entry** elements.

Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
    et/2021/maui"
3.             xmlns:x="http://schemas.microsoft.com/wi
        nfx/2009/xaml"

```

```

4.         x:Class="DataBindingSessionDemos.ObjectD
   ataBinding"
5.         Title="ObjectDataBinding">
6.     <VerticalStackLayout Spacing="25">
7.         <Entry Text="{Binding Name}"
8.             VerticalOptions="Center"
9.             HorizontalOptions="Center" />
10.        <Entry Text="{Binding Email}"
11.            VerticalOptions="Center"
12.            HorizontalOptions="Center" />
13.    </VerticalStackLayout>
14.</ContentPage>

```

In the code-behind file, all you have to do is to create the **Person** object and assign it to the **BindingContext** property of the UI. Refer to the following code:

```

1. public partial class Demo : ContentPage
2. {
3.     public ObjectDataBinding()
4.     {
5.         InitializeComponent();
6.         this.BindingContext = new Person { Name="John Doe",
7.             Email = "j.doe@msn.com" };
8.     }
9. }

```

In the above code snippet, you can see that the content in the code-behind file is much shorter, for the same functionality.

Note: Please note that this simple example was chosen to give a foundational understanding of XAML in data binding. The correct way is to extend the model class within the **INotifyPropertyChanged** interface. The following code is for your reference:

```

1. public class Person : INotifyPropertyChanged
2. {
3.     private string name;
4.     public string Name
5.     {

```

```

6.         get { return name; }
7.         set
8.         {
9.             if (name != value)
10.            {
11.                name = value;
12.                OnPropertyChanged("Name");
13.            }
14.        }
15.    }
16.    private string email;
17.    public string Email
18.    {
19.        get { return email; }
20.        set
21.        {
22.            if (email != value)
23.            {
24.                email = value;
25.                OnPropertyChanged("Email");
26.            }
27.        }
28.    }
29.    public event PropertyChangedEventHandler PropertyChanged;
30.    private void OnPropertyChanged(string propertyName)
31.    {
32.        if (this.PropertyChanged != null)
33.            this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
34.    }
35. }

```

We will discuss this in detail in the upcoming chapters when we use data binding in our projects.

Conclusion

In this chapter, you have learned how a .NET MAUI user interface is constructed like a tree. You can build this tree either by writing code in C# or using a language called XAML. We discussed some of the key points about XAML, including its syntax rules, attributes, markup extensions, namespaces, data binding, etc.

In the next chapter, we will apply the knowledge acquired so far to work on our first project and build a Color Picker app.

Points to remember

Here are some key takeaways from this chapter:

- XAML tree is a group of elements arranged like a family tree.
- There is always one element at the top of the tree called the root element.
- Each element in the tree corresponds to a class in .NET MAUI.
- Elements can have attributes which are just properties of the classes they represent.
- XAML syntax provides four types of constructs:
 - Object element syntax
 - Property-element syntax
 - Empty element syntax
 - Attached property syntax

Let us review the different XAML syntax forms that we discussed above:

Name	Description	Example
Object element syntax	Creates .NET MAUI object	<Button></Button>
Attribute syntax	Sets the properties of a .NET MAUI class object	<Button Text="Click"> </Button>

Empty element syntax	Creates .NET MAUI object without any content section	<Button Background="Red" />
Property element syntax	Sets the properties of a .NET AUI class object	<Button.Background> Red </Button.Background>
Attached property syntax	Sets an attached property on a .NET MAUI class object	<Button Grid.Row="1" />

CHAPTER 6

Project-1: Color Picker

Introduction

Now that we have the pre-requisite skills, it is time to get our hands dirty and build our first project. In this chapter, we will develop a Color Picker using the concepts we have learned so far. A Color Picker, also called RGB Color Mixer, is an essential tool for graphic designers, artists, and developers who need to select colors for their application. It is found in various graphic design software, image editing applications, and web development tools. The Color Picker allows users to adjust the values of red, green, and blue color components individually or in combination to create desired colors.

Structure

In this chapter, we will discuss the following topics:

- Creating the layout
- Creating the data entry section
- Specifying colors in XAML
- Changing the background color
- Generating random colors
- Copying the color to the clipboard

Objectives

After reading this chapter, you will be able to:

- Create a simple .NET MAUI app using commonly used controls.
- Develop the UI using XAML.
- Understand how to work with colors in XAML.
- You will also learn how to change the background color property of a control in your app, and generate random colors using C# in your app.

Creating the layout

Follow these steps to create a layout for the app:

1. Start Visual Studio.
2. In the dialog that appears, select the option Create a new project as shown in *Figure 6.1*:

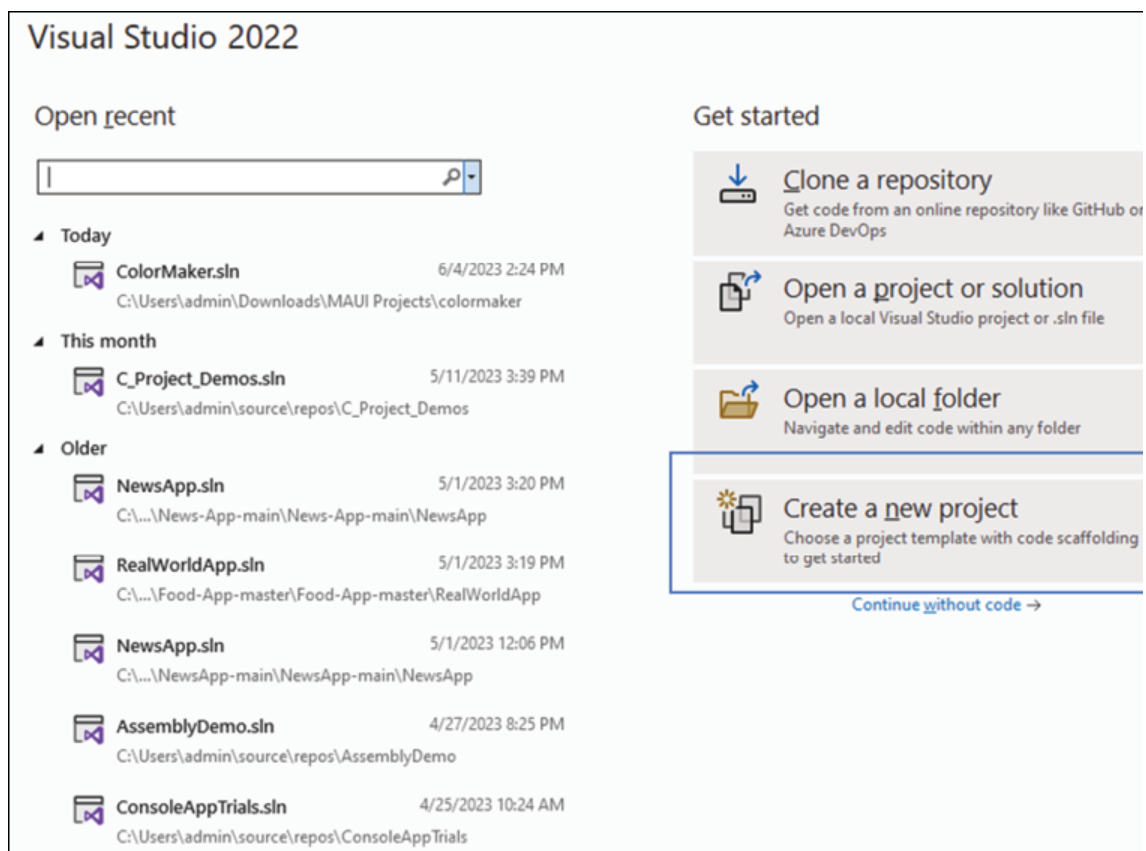


Figure 6.1: Create a new project in Visual Studio

3. Select the programming language as C# and MAUI project to view the MAUI project templates as shown in the following screenshot:

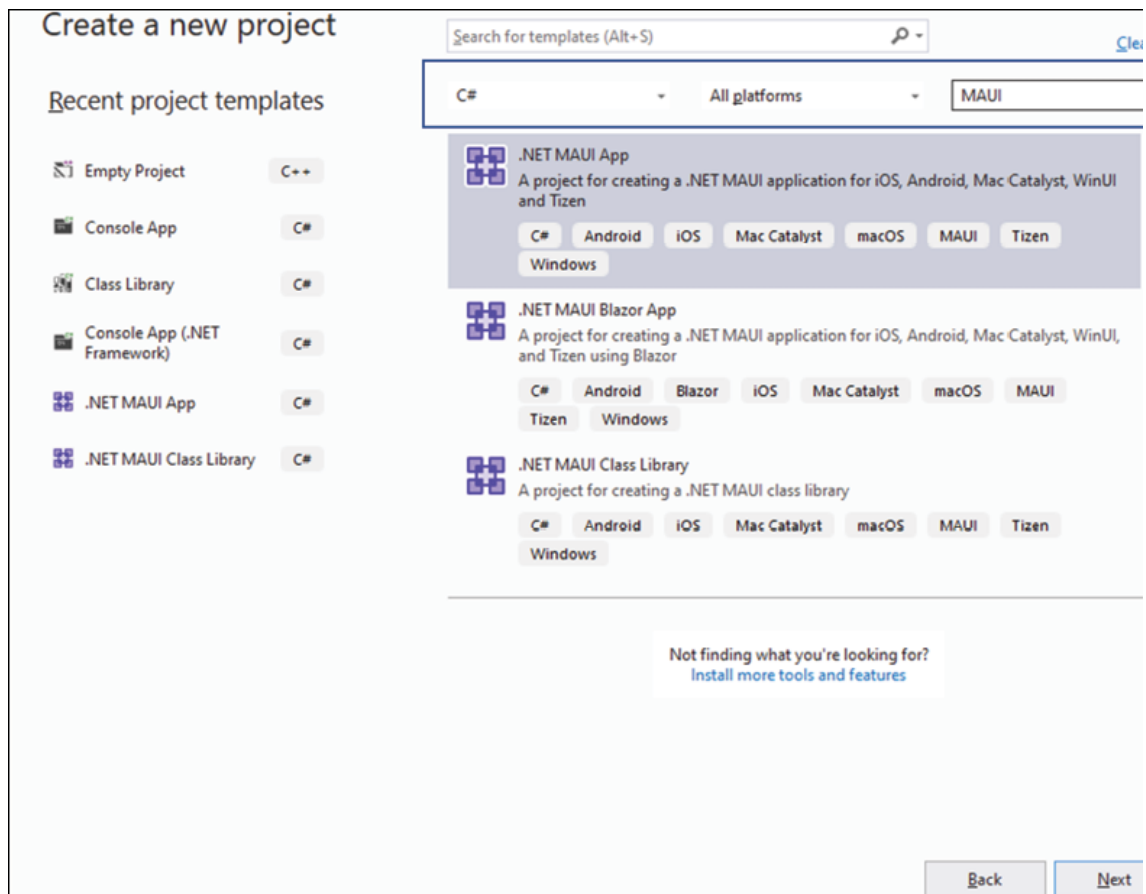


Figure 6.2: Select MAUI from the drop-down list under project types

4. Select .NET MAUI App and click on Next.
5. Enter the Project name as **ColorPicker**. Select the preferred destination folder or leave it with the default setting, as shown in the following figure:

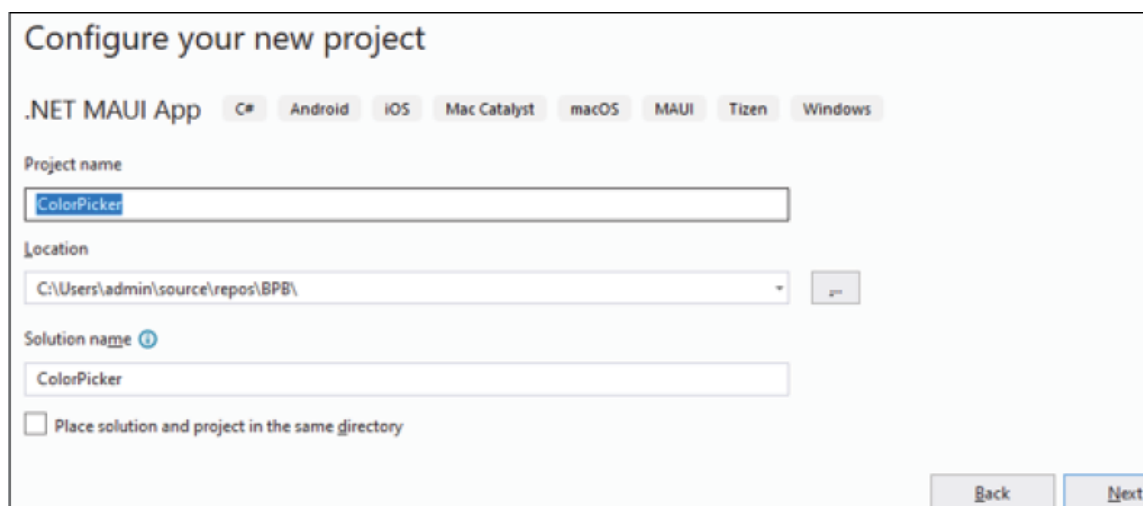


Figure 6.3: Configure your project

6. You can choose the latest Framework version, which is displayed under Additional information and click on Create:



Figure 6.4: Select the .NET Framework version

7. The MAUI project is created, and within a few minutes, you should be able to see the project in Solution Explorer:

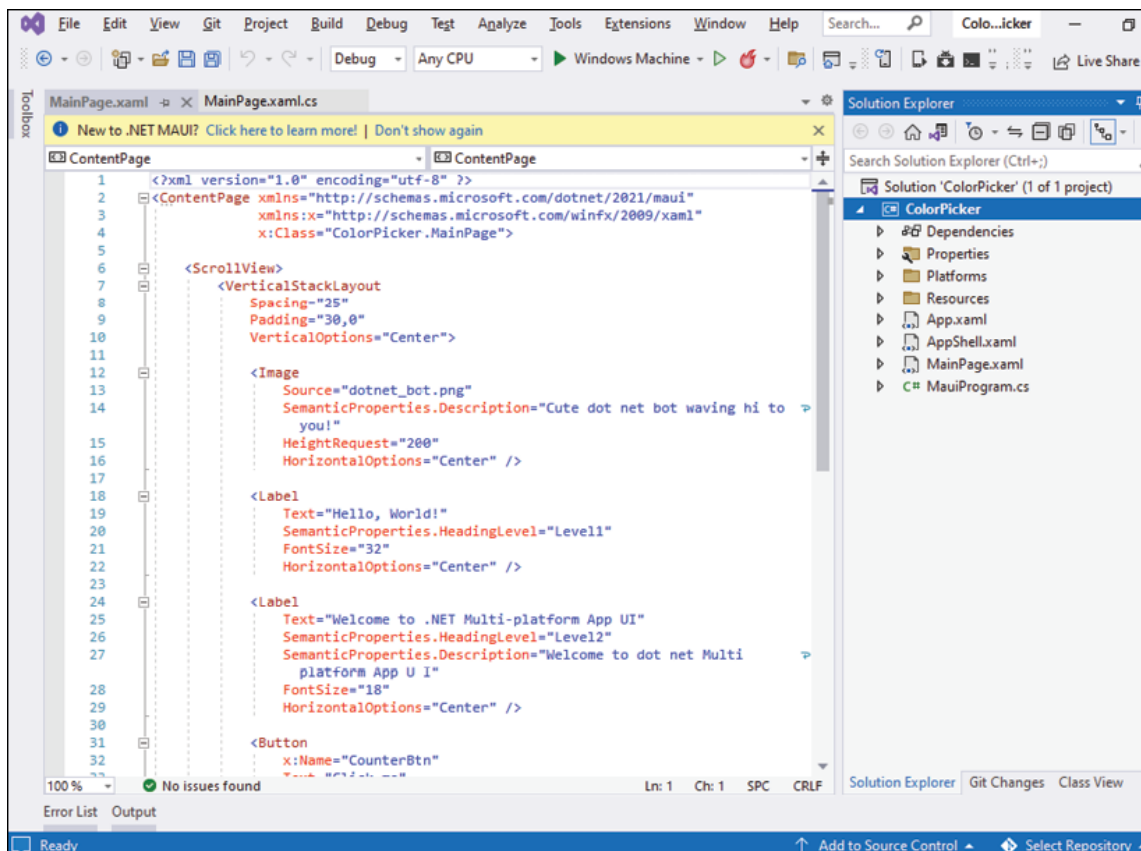


Figure 6.5: Default solution in Visual Studio

8. Make the following changes to the boilerplate code in **MainPage.xaml**. The **ContentPage** will serve as the parent for all the controls in your UI. Refer to the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>
```

```

2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
3.     xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
4.     x:Class="ColorPicker.MainPage">
5.
6.
7.
8. </ContentPage>

```

9. Open the code-behind file **MainPage.xaml.cs** and remove the default project code, as shown in the following code:

```

1. namespace ColorPicker;
2.
3. public partial class MainPage : ContentPage
4. {
5.
6.     public MainPage()
7.     {
8.         InitializeComponent();
9.     }
10.
11. }

```

10. Since this is a simple app with few controls, we will use a **Grid** which will serve as the app background. Assign a name to the **Grid** using the **x:Name** attribute, which will let us access the grid in the code-behind file. Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn

```

```

et/2021/maui"
3.         xmlns:x="http://schemas.microsoft.com/wi
nfx/2009/xaml"
4.         x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground"
7.         BackgroundColor="Black">
8.
9.     </Grid>
10.
11. </ContentPage>

```

Creating the data entry section

The color picker will need a few controls to set the red, green, and blue values. For each color component, we will need a Slider for adjusting the value and a Label to indicate which color it is for. Follow these steps to create the data entry section:

1. Let us define a frame that will serve as a layout with a border that will help us wrap all the other controls inside it. Inside the **Frame**, let us define a **VerticalStackLayout**, which will, in turn, hold the label-slider pairs for each of the red, green, and blue components of a color. Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
et/2021/maui"
3.         xmlns:x="http://schemas.microsoft.com/wi
nfx/2009/xaml"
4.         x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground"

```

```

7.         BackgroundColor="Black">
8.         <Frame>
9.             <VerticalStackLayout>
10.                <Label></Label>
11.                <Slider></Slider>
12.                <Label></Label>
13.                <Slider></Slider>
14.                <Label></Label>
15.                <Slider></Slider>
16.            </VerticalStackLayout>
17.        </Frame>
18.    </Grid>
19.
20.</ContentPage>

```

2. Assign the following values to the **Text** properties of the **Label** controls:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
3.     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4.     x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground"
7.         BackgroundColor="Black">
8.         <Frame>
9.             <VerticalStackLayout>

```

```

10.         <Label Text="Red Value:"></Label>
11.     <Slider></Slider>
12.         <Label Text="Green Value:"></Label>
13.     <Slider></Slider>
14.         <Label Text="Blue Value:"></Label>
15.     <Slider></Slider>
16. </VerticalStackLayout>
17. </Frame>
18. </Grid>
19.
20. </ContentPage>

```

3. Assign names to the **Slider** controls, as we will have to access these from the code-behind file. Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
3.         xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
4.         x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground"
7.         BackgroundColor="Black">
8.         <Frame>
9.             <VerticalStackLayout>
10.                 <Label Text="Red Value:"></Label>
11.                 <Slider x:Name="rValue"></Slider>
12.                 <Label Text="Green Value:"></Label>

```

```

13.         <Slider x:Name="gValue"></Slider>
14.         <Label Text="Blue Value:"></Label>
15.         <Slider x:Name="bValue"></Slider>
16.     </VerticalStackLayout>
17. </Frame>
18. </Grid>
19.
20. </ContentPage>

```

With the above markup, let us build and run our solution. You might see something like the details displayed in the following figure:



Figure 6.6: Running the app in emulator

4. The layout still needs improvement. You can use the **VerticalOptions** attribute in the **Frame** and **VerticalStackLayout** and also apply a **Spacing** of **20** to give a little bit of breathing space between the controls. Refer to the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
3.           xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
4.           x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground">
7.         <Frame Margin="10,0,10,0"
8.               VerticalOptions="Center">
9.             <VerticalStackLayout VerticalOptions="Cen
   ter"
10.                  Spacing="20">
11.                 <Label Text="Red Value:"></Label>
12.                 <Slider x:Name="rValue"></Slider>
13.                 <Label Text="Green Value:"></Label>
14.                 <Slider x:Name="gValue"></Slider>
15.                 <Label Text="Blue Value:"></Label>
16.                 <Slider x:Name="bValue"></Slider>
17.             </VerticalStackLayout>
18.         </Frame>
19.     </Grid>
20.
```

21. `</ContentPage>`

5. After making these changes, you can build and run the app. You should be able to see a better-looking screen like the following figure:

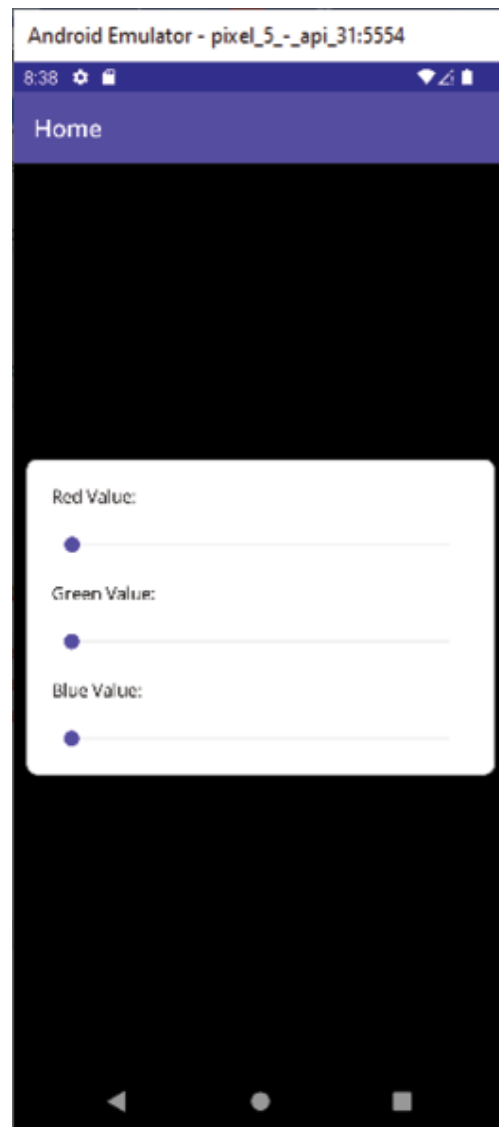


Figure 6.7: Improved layout of the app

Specifying colors in XAML

In XAML, colors can be set in two ways:

- Using one of the 148 named colors like black, blue, red, etc.
- Using the hexadecimal notation, prefixed with a # symbol.

Earlier, we had set the background color of the grid to black. This could have been done alternatively using the **Hex** notation like the following command:

1. `<Grid x:Name="AppBackground"`
2. `BackgroundColor="#000000">`

When using the **Hex** notation, the string is parsed in the following order:



Figure 6.8: Hex notation for color

Hence, to apply a red color to the background, you can use the following markup:

3. `<Grid x:Name="AppBackground"`
4. `BackgroundColor="#FF0000">`

If you run the app, it will display a red colored background, as seen in the following figure:

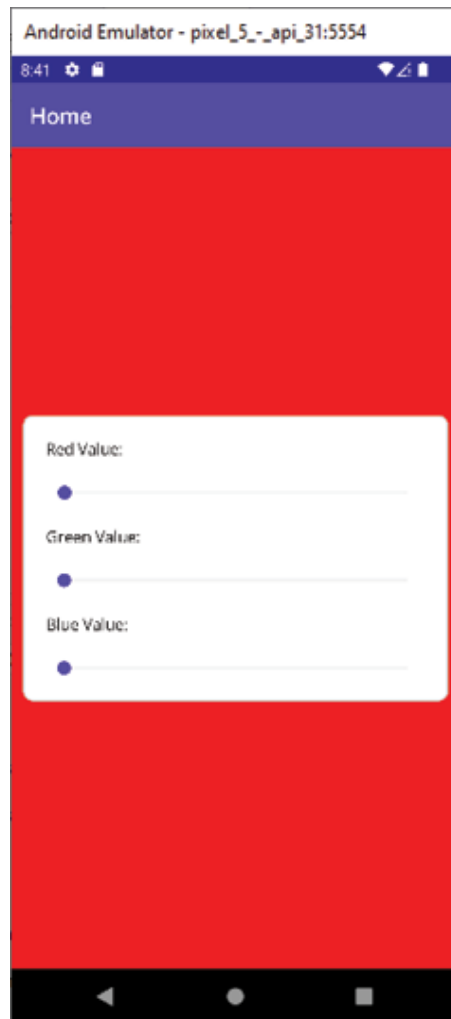


Figure 6.9: Layout after applying background color

Alternatively, you can also set the background color from the code-behind file. The equivalent C# code is shown in line number 14. As you can see, the **Color** class is in **Microsoft.Maui.Graphics** namespace, exposes a static method **FromRgb()** which lets you specify the red-green-blue values in the arguments. Refer to the following code:

```

1. namespace ColorPicker;
2.
3. public partial class MainPage : ContentPage
4. {
5.
6.     public MainPage()
7.     {
8.         InitializeComponent();
9.         this.Loaded += MainPage_Loaded;
10.    }
11.

```

```

12. private void MainPage_Loaded(object sender, EventArgs e
    )
13. {
14.     AppBackground.BackgroundColor = Color.FromRgb(255, 0,
        0);
15. }
16. }

```

Try changing the argument values passed to the static method **Color.FromRgb()** to display different colors like the following figure:

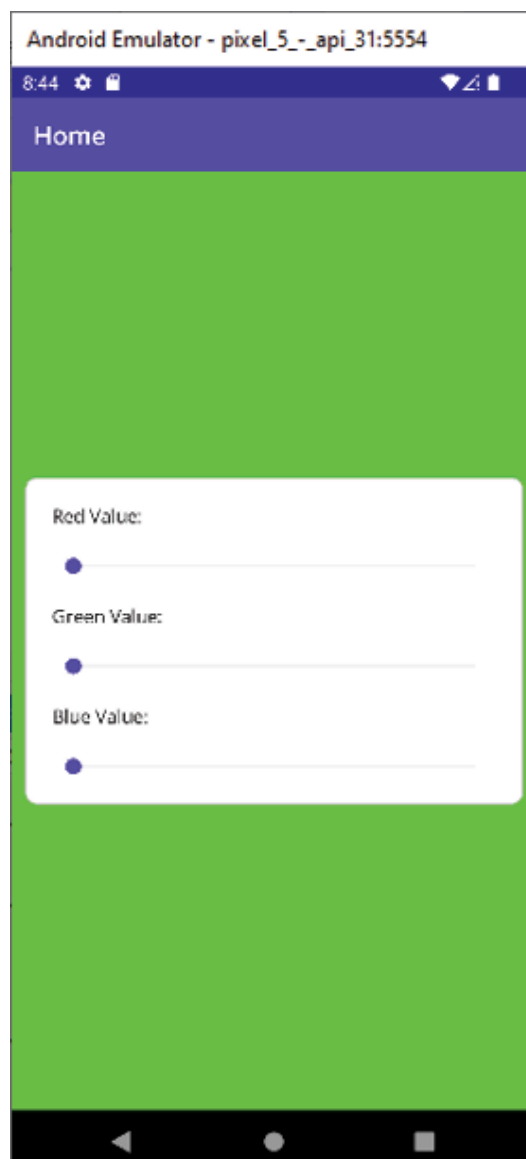


Figure 6.10: Changing the background color from code-behind file

Remember that whenever you make changes in XAML you do not have to stop debugging and start again. XAML live load feature will update the UI in

your emulator as soon as you save the changes. However, whenever you change the code in C#, you will have to stop debugging and start again.

Changing the background color

Now, that we understand how we can change the color from C#, let us now read the slider value and pass it as arguments to the **Color.FromRgb()** method. The **Slider** defines a **ValueChanged** event, which is raised whenever the value changes when the user manipulates the slider. The following code is the code behind the file contains the handler for the **ValueChanged** event:

```
1. namespace ColorPicker;
2.
3. public partial class MainPage : ContentPage
4. {
5.
6.     public MainPage()
7.     {
8.         InitializeComponent();
9.         this.Loaded += MainPage_Loaded;
10.    }
11.
12.    private void MainPage_Loaded(object sender, EventArgs e
13.    )
14.    {
15.        AppBackground.BackgroundColor = Color.FromRgb(0, 255,
16.        0);
17.    }
18.
19.    private void slrValue_ValueChanged(object sender, Value
20.    ChangedEventArgs e)
21.    {
22.
23.        var red = rValue.Value;
24.        var green = gValue.Value;
25.        var blue = bValue.Value;
```

```
lor);  
26.    }  
27. }
```

Now, run the application and change the values of the sliders for red, green, and blue values to see the background color of the grid changing immediately, as shown in the following screenshot:

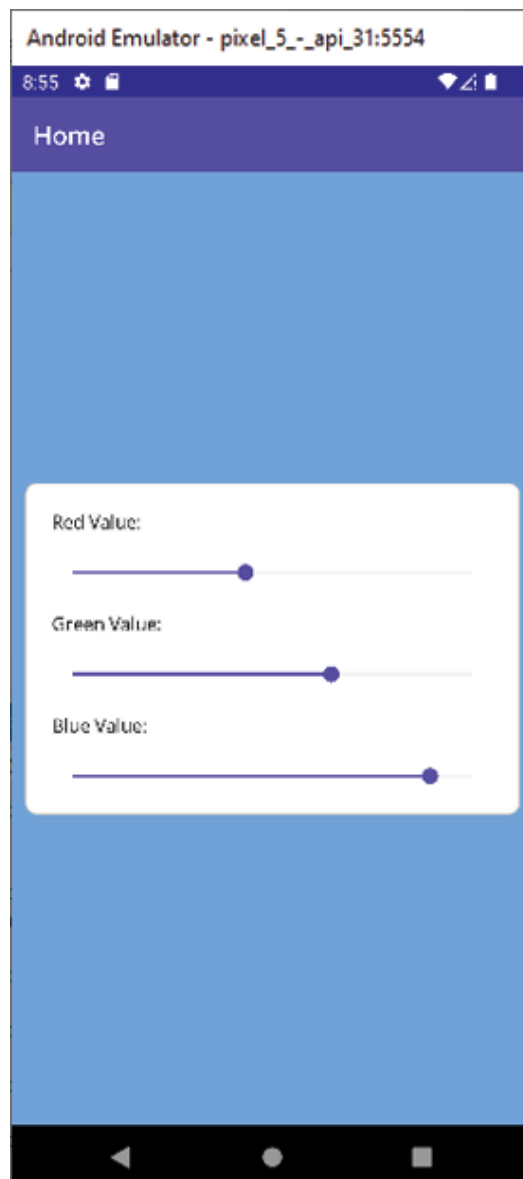


Figure 6.11: Changing the background color by adjusting the slider controls

Generating random colors

Now, let us implement the functionality to generate random colors in the app. Add a button in the UI as shown in lines 20 and 21, and assign an event

handler for the **Clicked** event:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/
   2021/maui"
3.           xmlns:x="http://schemas.microsoft.com/winfx
   /2009/xaml"
4.           x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground">
7.         <Frame Margin="10,0,10,0"
8.             VerticalOptions="Center">
9.             <VerticalStackLayout VerticalOptions="Center
   "
10.                Spacing="20">
11.                 <Label Text="Red Value:"></Label>
12.                 <Slider x:Name="rValue"
13.                     ValueChanged="slrValue_ValueChan
   ged"></Slider>
14.                 <Label Text="Green Value:"></Label>
15.                 <Slider x:Name="gValue"
16.                     ValueChanged="slrValue_ValueChan
   ged"></Slider>
17.                 <Label Text="Blue Value:"></Label>
18.                 <Slider x:Name="bValue"
19.                     ValueChanged="slrValue_ValueChan
   ged"></Slider>
20.                 <Button Text="Generate Random Color"
21.                     Clicked="Button_Clicked">
22.                 </Button>
23.             </VerticalStackLayout>
24.         </Frame>
25.     </Grid>
26. </ContentPage>
```

In the code-behind file, implement the **Clicked** event handler as shown in the following code. We are using the **Random** class provided in .NET to generate a random number between the two ranges which are passed as arguments to the **random.Next()** method. Remember that RGB values can

take a maximum value of FF in hexadecimal which is 255 in decimal notation. Refer to the following code:

```
1. using Android.Content.Res;
2. using Kotlin.Jvm.Internal;
3. using static Android.Graphics.BlurMaskFilter;
4.
5. namespace ColorPicker;
6.
7. public partial class MainPage : ContentPage
8. {
9.
10. public MainPage()
11. {
12.     InitializeComponent();
13.     this.Loaded += MainPage_Loaded;
14. }
15.
16. private void MainPage_Loaded(object sender, EventArgs e
    )
17. {
18.     AppBackground.BackgroundColor = Color.FromRgb(0, 255,
        0);
19. }
20.
21. private void slrValue_ValueChanged(object sender, Value
    ChangedEventArgs e)
22. {
23.     var red = rValue.Value;
24.     var green = gValue.Value;
25.     var blue = bValue.Value;
26.
27.     Color color = Color.FromRgb(red, green, blue);
28.
29.     AppBackground.BackgroundColor = color;
30. }
31.
32. private void Button_Clicked(object sender, EventArgs e)
33. {
```

```

34.         var random = new Random();
35.
36.         var color = Color.FromRgb(
37.             random.Next(0, 255),
38.             random.Next(0, 255),
39.             random.Next(0, 255));
40.
41.         AppBackground.BackgroundColor = color;
42.     }
43. }

```

Run the application now and click the Generate Random Color button to view random colors being set to the Grid's background, as shown in the following figure:

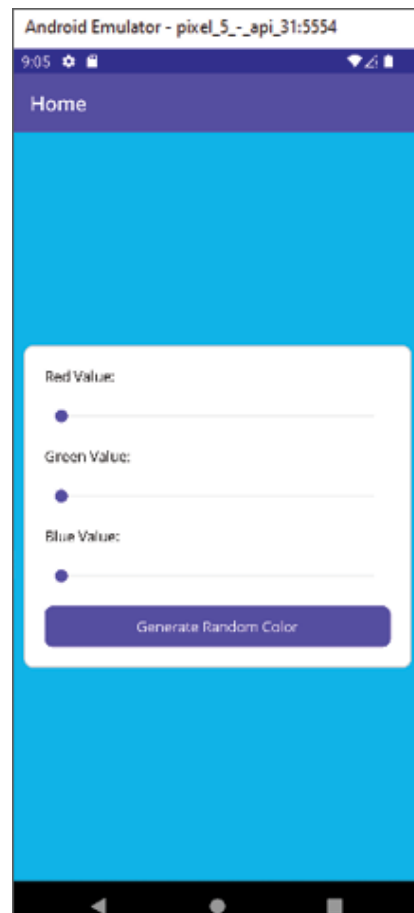


Figure 6.12: Added button to generate random color

Copying the color to the clipboard

The next feature that we will implement is the ability to copy the background color to the clipboard, generated either through manipulation of the sliders or by clicking the button.

First, we will provide a label to display the value of the color that is generated and provide a button to copy this value to the clipboard. In order to do that, we have added a **Frame** below all the controls and provided a **HorizontalStackLayout** inside it, which will host the **Label** and the **ImageButton** side by side. The **Text** shown in the **Label** must be assigned from C# and hence provide it with a name. The **ImageButton** is similar to a button except that it can display an image as well. Use an SVG image saved to the **Resources** folder of your project to assign to the **ImageButton**'s Source property.

You can download SVG icons for free from many websites such as https://www.iconfinder.com/icons/211649/clipboard_icon, which is used in our example:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/
   2021/maui"
3.             xmlns:x="http://schemas.microsoft.com/winfx
   /2009/xaml"
4.             x:Class="ColorPicker.MainPage">
5.
6.     <Grid x:Name="AppBackground">
7.         <Frame Margin="10,0,10,0"
8.             VerticalOptions="Center">
9.             <VerticalStackLayout VerticalOptions="Center
   "
10.                Spacing="20">
11.                 <Label Text="Red Value:"></Label>
12.                 <Slider x:Name="rValue"
13.                     ValueChanged="slrValue_ValueChan
   ged"></Slider>
14.                 <Label Text="Green Value:"></Label>
15.                 <Slider x:Name="gValue"
16.                     ValueChanged="slrValue_ValueChan
   ged"></Slider>
17.                 <Label Text="Blue Value:"></Label>
18.                 <Slider x:Name="bValue"
```

```

19.         ValueChanged="slrValue_ValueChan
ged"></Slider>
20.         <Button Text="Generate Random Color"
21.             Clicked="Button_Clicked">
22.         </Button>
23.         <Frame HorizontalOptions="Center">
24.             <HorizontalStackLayout>
25.                 <Label x:Name="rgbValue"
26.                     VerticalOptions="Center">
27.                 </Label>
28.                 <ImageButton x:Name="copyBtn"
29.                     Source="clipboard_i
con.svg"
30.                     HeightRequest="25"
31.                     WidthRequest="25"
32.                     Margin="10,0,0,0"
33.                     Clicked="copyBtn_Cl
icked"></ImageButton>
34.             </HorizontalStackLayout>
35.         </Frame>
36.     </VerticalStackLayout>
37. </Frame>
38. </Grid>
39. </ContentPage>

```

In the **ImageButton**'s Clicked event handler, implement the code to copy to the *Clipboard*. Note that we have optimized the program to avoid repetitive statements and created a method **SetBackgroundColor()** which takes a **Color** object as an argument. In addition to setting the **Background** color of the grid, this method also converts the color value to a string which is displayed in the newly added **Label** in our UI. Refer to the following code:

```

1. using Android.Content.Res;
2. using Kotlin.Jvm.Internal;
3. using static Android.Graphics.BlurMaskFilter;
4.
5. namespace ColorPicker;
6.
7. public partial class MainPage : ContentPage

```

```

8. {
9.
10. public MainPage()
11. {
12.     InitializeComponent();
13.     this.Loaded += MainPage_Loaded;
14. }
15.
16. private void MainPage_Loaded(object sender, EventArgs e)
17. {
18.     Color color = Color.FromRgb(0,0,0);
19.     SetBackgroundColor(color);
20. }
21.
22. private void slrValue_ValueChanged(object sender, ValueChangedEventArgs e)
23. {
24.     var red = rValue.Value;
25.     var green = gValue.Value;
26.     var blue = bValue.Value;
27.
28.     Color color = Color.FromRgb(red, green, blue);
29.
30.     SetBackgroundColor(color);
31. }
32.
33. private void Button_Clicked(object sender, EventArgs e)
34. {
35.     var random = new Random();
36.
37.     var color = Color.FromRgb(
38.         random.Next(0, 255),
39.         random.Next(0, 255),
40.         random.Next(0, 255));
41.
42.     SetBackgroundColor(color);
43. }

```

```

44.
45.     private void SetBackgroundColor(Color color)
46.     {
47.         AppBackground.BackgroundColor = color;
48.         rgbValue.Text = color.ToHex();
49.     }
50.
51.     private async void copyBtn_Clicked(object sender, EventArgs e)
52.     {
53.         await Clipboard.SetTextAsync(rgbValue.Text);
54.     }
55. }

```

When the user clicks on the **ImageButton** and copies it to the clipboard, it is also important to give feedback to the user. For this purpose, let us include one more Label named **lblInfo** in our user interface. This Label can be used to provide feedback to the user when the color is copied to the clipboard, as shown in the following code:

```

1. namespace ColorPicker;
2.
3. public partial class MainPage : ContentPage
4. {
5.
6.     public MainPage()
7.     {
8.         InitializeComponent();
9.         this.Loaded += MainPage_Loaded;
10.    }
11.
12.    private void MainPage_Loaded(object sender, EventArgs e)
13.    {
14.        Color color = Color.FromRgba(0,0,0,0);
15.        SetBackgroundColor(color);
16.    }
17.
18.    private void slrValue_ValueChanged(object sender, ValueChangedEventArgs e)

```

```

19. {
20.     var red = rValue.Value;
21.     var green = gValue.Value;
22.     var blue = bValue.Value;
23.
24.     Color color = Color.FromRgb(red, green, blue);
25.
26.     SetBackgroundColor(color);
27. }
28.
29. private void Button_Clicked(object sender, EventArgs e)
30. {
31.     var random = new Random();
32.
33.     var color = Color.FromRgb(
34.         random.Next(0, 255),
35.         random.Next(0, 255),
36.         random.Next(0, 255));
37.
38.     SetBackgroundColor(color);
39. }
40.
41. private void SetBackgroundColor(Color color)
42. {
43.     lblInfo.Text = "";
44.     AppBackground.BackgroundColor = color;
45.     rgbValue.Text = color.ToHex();
46. }
47.
48. private async void copyBtn_Clicked(object sender, EventArgs e)
49. {
50.     await Clipboard.SetTextAsync(rgbValue.Text);
51.     lblInfo.Text = "Copied to clipboard";
52. }
53. }
54.

```

Run the app after making the above changes, and when you click **ImageButton**, the text should be displayed as shown in the following figure:

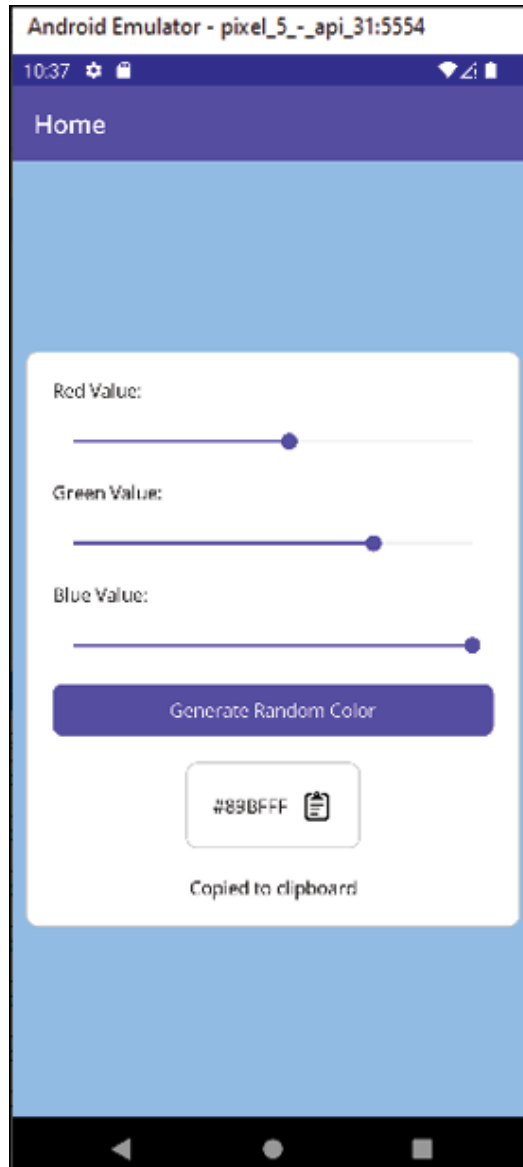


Figure 6.13: Copying the random color to the clipboard

Now that our app is complete, we have one more change to do. Notice that the **Header** in the UI has been displaying the default text **Home**. Edit the **AppShell.xaml** file and replace the Title attribute of **ShellContent** class with the string **Color Picker** or your preferred title. Refer to the following code:

1. `<?xml version="1.0" encoding="UTF-8" ?>`
2. `<Shell`
3. `x:Class="ColorPicker.AppShell"`
4. `xmlns="http://schemas.microsoft.com/dotnet/2021/maui"`


```
"
5.     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
6.     xmlns:local="clr-namespace:ColorPicker"
7.     Shell.FlyoutBehavior="Disabled">
8.
9.     <ShellContent
10.         Title="Color Picker"
11.         ContentTemplate="{DataTemplate local:MainPage}"
12.         Route="MainPage" />
13.
14. </Shell>
```

Run the app after making the above change, and you should see the final app view as shown in the following figure:

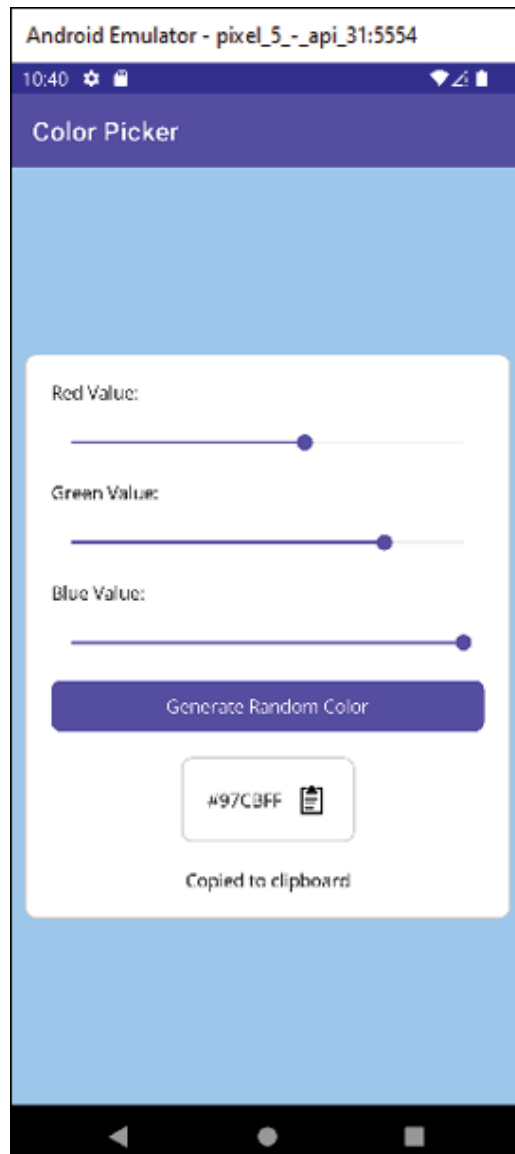


Figure 6.14: App User Interface after updating the App Name

Conclusion

In this chapter, you have developed your first project using .NET MAUI, and hopefully, you have been able to run it on your device as well. In case you faced any errors, please check the source code provided for reference, compare it with your code and make the necessary corrections.

In the next chapter, we will improve the skills further by developing a Tip Calculator.

Points to remember

Here are some key takeaways from this chapter:

- Ensure to choose the correct .NET Framework version while creating the project.
- This .NET MAUI app consists of a hierarchy of controls starting with the `ContentPage`, which in turn holds a `Grid`. The `Grid` control holds the `Frame`, and the `Frame`, in turn, holds the `VerticalStackLayout`. This is only to demonstrate how you can develop a polished app easily. If you prefer, you can also develop the UI using other arrangement of these controls.
- We used a solid color to assign to the `BackgroundColor` property. You can also use Gradients to improve the aesthetics of the app.
- If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Project-2: Tip Calculator

Introduction

In this chapter, you will be introduced to creating a Tip Calculator. A Tip Calculator is a tool or app that helps you figure out how much tip to leave after having food at a restaurant. The tip amount varies depending on the country and cultural norms. In many countries, it is customary to leave a tip of around 15% to 20% of the total bill at restaurants. However, some places may have different tipping customs, and some people may choose to leave more or less than the usual percentage based on the quality of service or personal preferences. Also, when dining in a group, it is common to split the total bill, including the tip, equally among all the individuals to ensure a fair distribution of expenses. This method allows everyone to contribute their share, making it easier to handle the bill and the tip without any confusion or inconvenience. In this chapter, you will learn how to build a Tip Calculator that will allow users to enter the bill amount, select the desired tip percentage, and the number of people who will split the bill. It will help you instantly calculate the tip amount and the split amount for each user from the total bill amount.

Structure

In this chapter, we will discuss the following topics:

- Creating the layout
- Creating the data entry section
- Applying graphical design principles in UI design

- Performing the calculation

Objectives

After reading this chapter, you will be able:

- To create a complex UI layout using .NET MAUI controls such as Grid, Frame, VerticalStackLayout, and HorizontalStackLayout controls.
- You will also be able to develop the UI in XAML. Further, you will understand how to work with commonly used controls, including Label, Entry, Slider, and Button.
- You will know how to implement the functionality to calculate the tip amount and split the amount in C#.

Creating the layout

Follow these steps to create a layout for the app:

1. Start Visual Studio.
2. In the dialog that appears, select the option: **Create a new project**, as shown in [Figure 7.1](#):

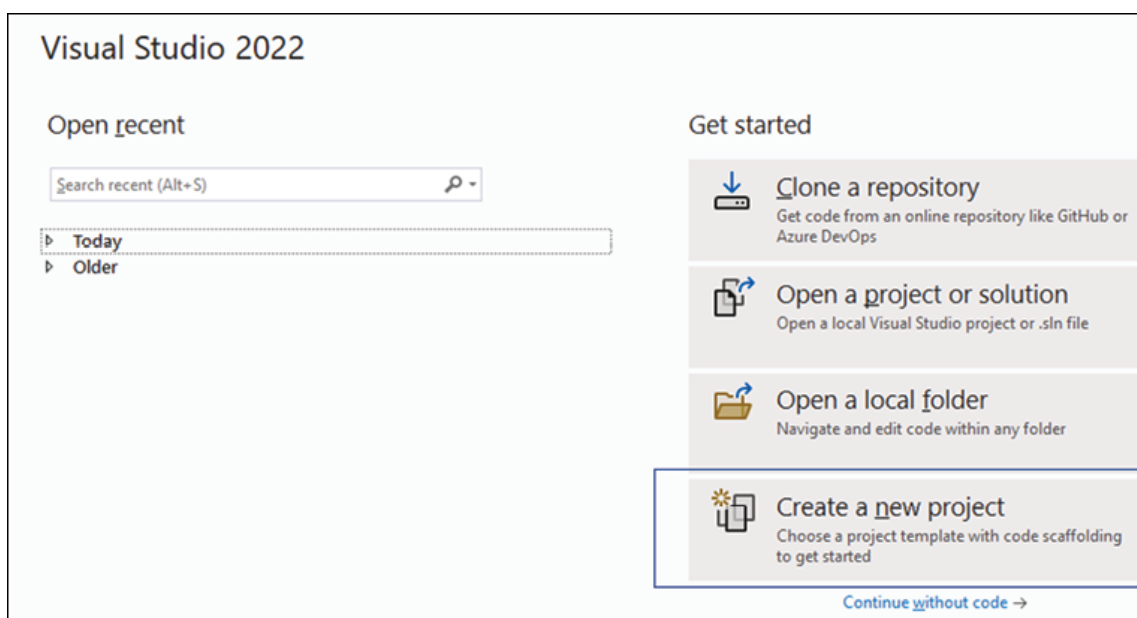


Figure 7.1: Creating a new project in Visual Studio

3. Select the programming language as C# and MAUI project to view the MAUI project templates, as shown in [Figure 7.2](#):

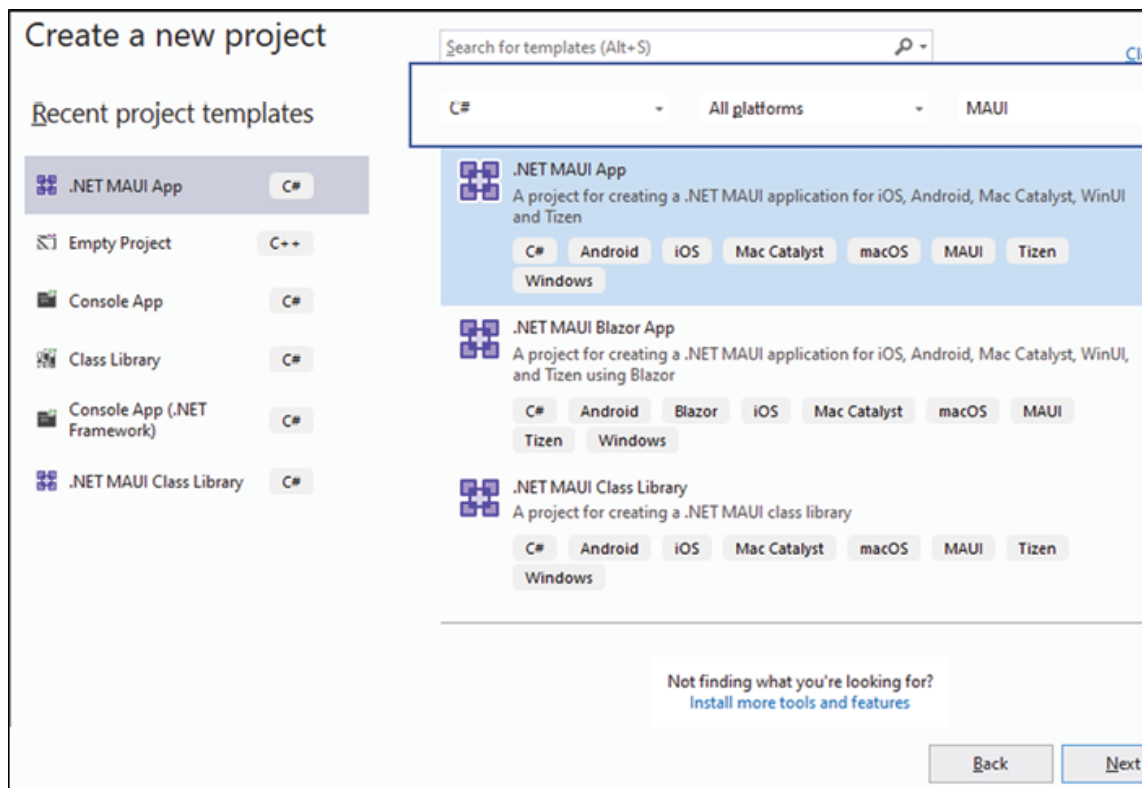


Figure 7.2: Selecting .NET MAUI project template

4. Select **.NET MAUI App** and click on **Next**.
5. Enter the Project name as **TipCalculator**.
6. Select the preferred destination folder or leave it with the default setting as shown in [Figure 7.3](#):

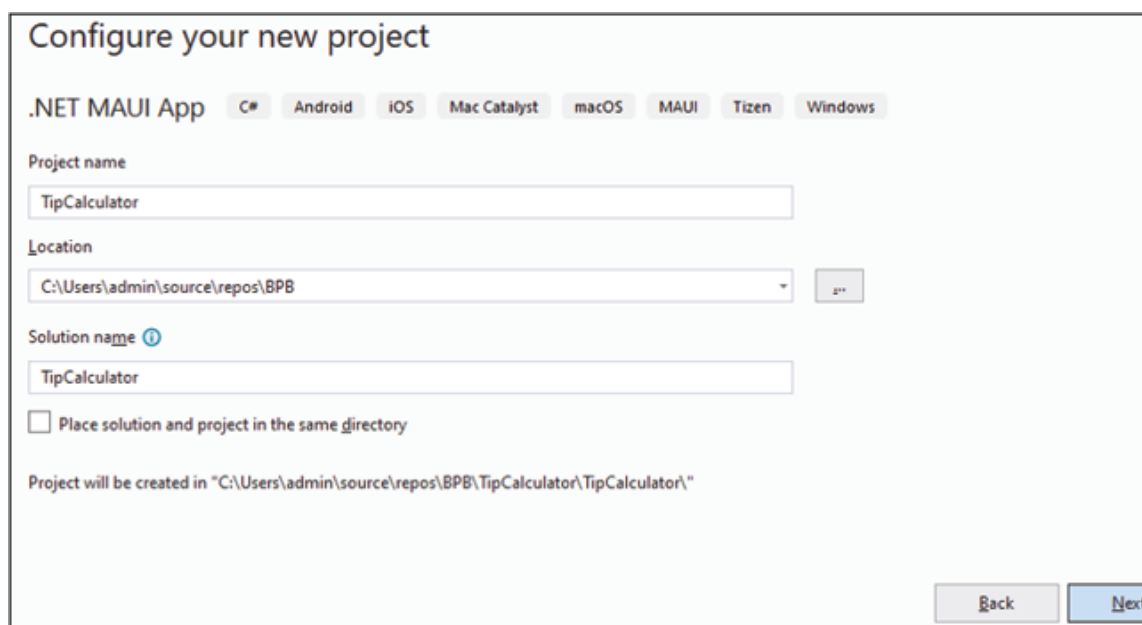


Figure 7.3: Configure your project

7. You can choose the latest **Framework** version, which is displayed under **Additional information**, and click on **Create**, as shown in the following figure:

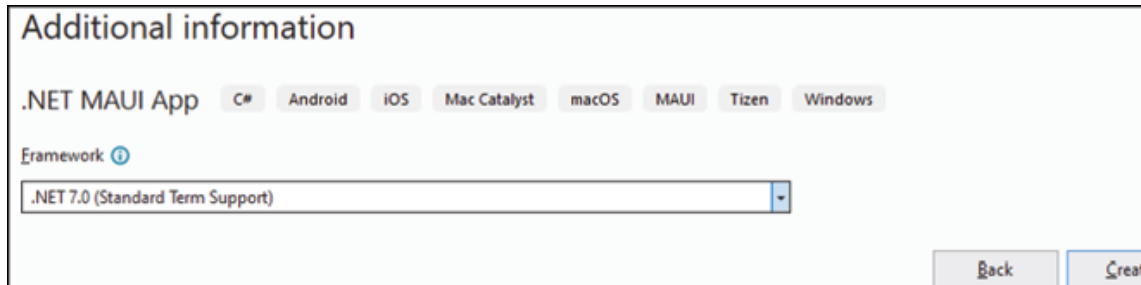


Figure 7.4: Selecting the framework version

8. The MAUI project is created, and within a few minutes, you should be able to see the project in the **Solution Explorer**, as shown in the following figure:

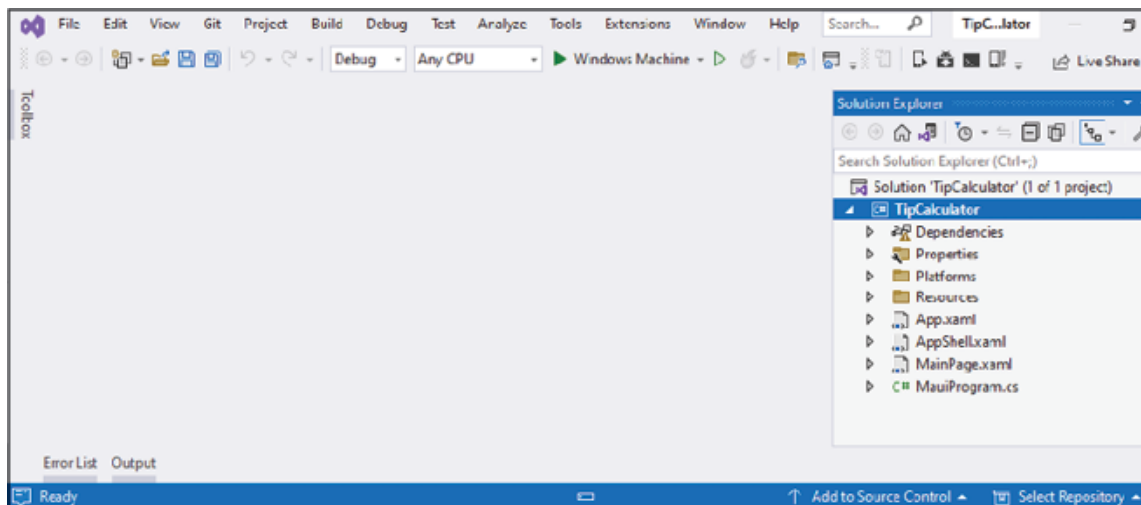


Figure 7.5: Default solution in Visual Studio

9. Delete the code between the **ContentPage** tags in the boilerplate code in **MainPage.xaml**. The **ContentPage** will serve as the parent for all the controls in your UI as shown in the following figure:

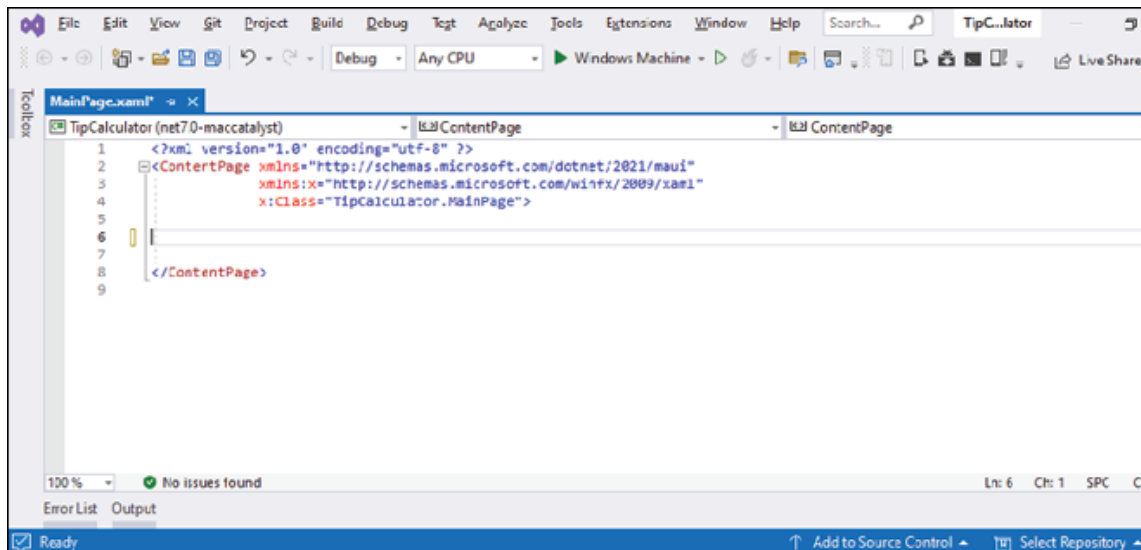


Figure 7.6: XAML markup after removing the default project code

10. Open the code-behind file **MainPage.xaml.cs** and remove the default project code, as shown in the following figure:

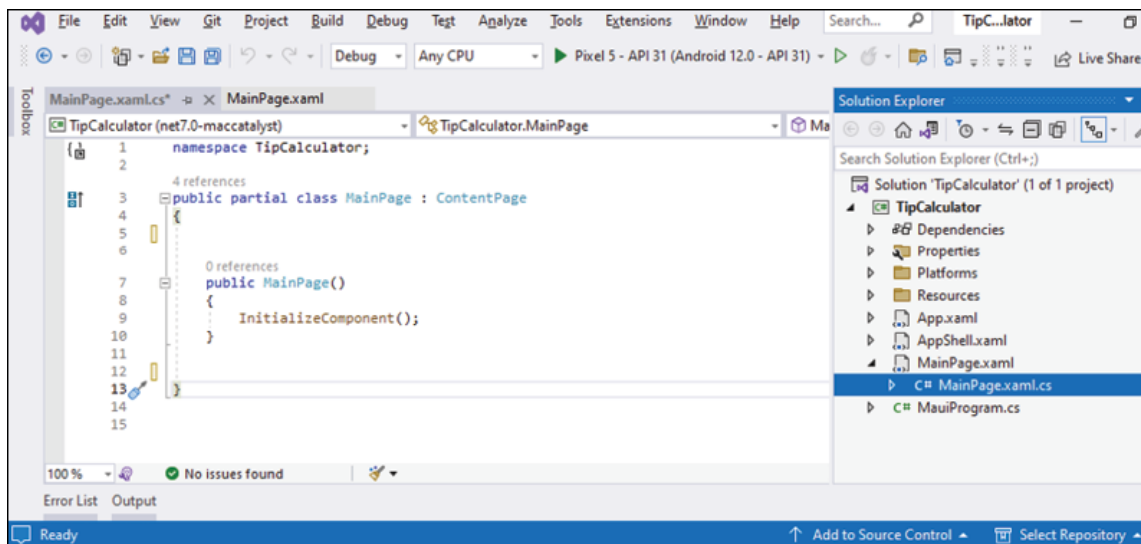


Figure 7.7: Code-behind file after removing the default project code

11. For this app, we are going to have a grid with multiple rows and columns which will serve as the container for the data entry controls that we will add next.
12. By default, a grid will be created with one row and one column. To create multiple rows and columns, we use the **Grid.RowDefinitions** and **Grid.ColumnDefinitions** properties of the **Grid**. Each **RowDefinition** declared inside the **Grid.RowDefinitions** creates a row, and similarly, each **ColumnDefinition** declared inside **ColumnDefinitions** creates a column. For our app, we will need 5 rows and 2 columns.

13. The height of the rows and width of the columns can be defined in three ways:
- Fixed value:** To assign a fixed size of logical units.
 - Auto:** To take up only as much space as is required for the controls in that specific row/column.
 - Star:** Use star notation for proportional sizing to specify sizes relative to each other based on proportions or percentages. This allows us to allocate available space dynamically among the rows or columns based on their specified proportions, which is better for creating flexible and responsive layouts.
14. We will use the proportional sizing approach to define the heights of the rows and width of the columns. Since the height of each row in the Grid must be different depending on the controls that we will be placing inside the respective rows, we will define the proportions accordingly.
15. The two columns can be of equal width and hence can be declared to have equal proportions. Hence, let us define the Grid's **RowDefinitions** and **ColumnDefinitions** accordingly. Refer to the following code:

```
1. <Grid Padding="5">
2.     <Grid.RowDefinitions>
3.         <RowDefinition Height="0.25*">
4.     </RowDefinition>
5.         <RowDefinition Height="0.25*">
6.     </RowDefinition>
7.         <RowDefinition Height="0.125*">
8.     </RowDefinition>
9.         <RowDefinition Height="0.25*">
10.    </RowDefinition>
11.     </Grid.RowDefinitions>
12.     <Grid.ColumnDefinitions>
13.         <ColumnDefinition Width="0.5*">
```

```

    </ColumnDefinition>
11.         <ColumnDefinition Width="0.5*">
    </ColumnDefinition>
12.     </Grid.ColumnDefinitions>
13. </Grid>

```

16. Any control can be placed within a **Grid** by using its **Grid.Row** and **Grid.Column** properties that represent which column and which row the control will be placed in. The values of rows and columns start with 0. That means, if there are two columns in the grid, the first column will be represented by the value **Grid.Column="0"** and the second column by the value **Grid.Column="1"**. Row numbers are also determined in a similar way.

17. For our app, we will place **Frame** controls inside the rows and columns. A Frame control is used to wrap a view or layout with a border that can be configured with color, shadow, and other options. Frame controls are generally used to create borders around controls but can also be used to create more complex UI. It may be worth mentioning here that a border can also be used in place of a Frame.

18. For each frame, we will provide a background color using the background property.

19. The frame controls in rows 1, 3 and 5 will have to stretch across the screen spanning two columns. Set the **Grid.ColumnSpan** property of the **Frame** control to 2, so that it stretches across two columns. Refer to the following code:

```

1.     <Grid Padding="5">
2.         <Grid.RowDefinitions>
3.             <RowDefinition Height="0.25*">
    </RowDefinition>
4.             <RowDefinition Height="0.25*">
    </RowDefinition>
5.             <RowDefinition Height="0.125*">
    </RowDefinition>
6.             <RowDefinition Height="0.25*">
    </RowDefinition>

```

```

7.         <RowDefinition Height="0.125*">
</RowDefinition>
8.     </Grid.RowDefinitions>
9.     <Grid.ColumnDefinitions>
10.         <ColumnDefinition Width="0.5*">
</ColumnDefinition>
11.         <ColumnDefinition Width="0.5*">
</ColumnDefinition>
12.     </Grid.ColumnDefinitions>
13.     <Frame Background="LightBlue"
14.         Grid.ColumnSpan="2"></Frame>
15.     <Frame Background="LightSteelBlue"
16.         Grid.Row="1"></Frame>
17.     <Frame Background="LightGreen"
18.         Grid.Row="1"
19.         Grid.Column="2"></Frame>
20.     <Frame Background="Beige"
21.         Grid.Row="2"
22.         Grid.ColumnSpan="2"></Frame>
23.     <Frame Background="LightSteelBlue"
24.         Grid.Row="3"></Frame>
25.     <Frame Background="LightGreen"
26.         Grid.Row="3"
27.         Grid.Column="2"></Frame>
28.     <Frame Background="Beige"
29.         Grid.Row="4"
30.         Grid.ColumnSpan="2"></Frame>

```

31. `</Grid>`

20. At this stage, the UI should look like the illustration in *Figure 7.8*:

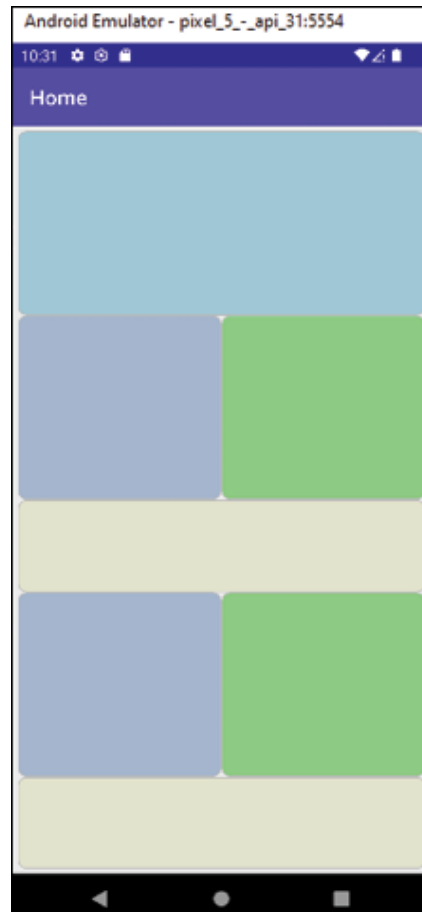


Figure 7.8: Layout of the app

Creating the data entry section

Follow these steps to insert the necessary controls to develop the UI that forms the data entry section:

1. In the previous section, we added **Frames** inside the **Grid**. As the **Frame** can take only one child, we will use a layout container like **VerticalStackLayout** or **HorizontalStackLayout** as the child of the **Frame** and then place the other controls of the UI inside it.
2. To position the child controls of the **VerticalStackLayout** in the center, you can set the two properties **HorizontalOptions=Center** and **VerticalOptions =Center**. However, we need to declare this for all the remaining **VerticalStackLayout** controls that we will be adding to the UI later. Therefore, it is advisable to define a **Style** and reuse it.

3. A style can be defined at any level of a page hierarchy, and all the descendant controls can use the style. For our app, we will declare it as a resource at the **Grid** level so that it is available for all the children of the **Grid**.
4. When declaring style, two properties are mandatorily required:
 - a. **TargetType** property, which indicates which type of control can use this **Style**.
 - b. **X:Key** property provides a name for the style.
5. You can define multiple properties inside a style using **Setter** element. The **Setter** element requires two attributes to be set, **Property** and **Value**.
6. Once you have defined the style, you can apply the style in the target control using the markup extension syntax as shown. Remember that the curly braces are mandatory here. Refer to the following code:

```

1.      <Grid.Resources>
2.          <Style TargetType="Frame"
3.              x:Key="centerFrame">
4.              <Setter Property="HorizontalOptions"
5.                  Value="Center" />
6.              <Setter Property="VerticalOptions"
7.                  Value="Center" />
8.          </Style>
9.      </Grid.Resources>
10.     <Frame Background="LightBlue"
11.         Grid.ColumnSpan="2">
12.         <VerticalStackLayout Style=
13.             "{StaticResource centerFrame}">
14.             </VerticalStackLayout>
15.         </Frame>

```

7. For the first row, add a **Label** and an **Entry** control for the bill amount, as shown in the following code:

```

1. <Grid.Resources>
2.     <Style TargetType="Frame"
3.         x:Key="centerFrame">
4.         <Setter Property="HorizontalOptions"
5.             Value="Center" />
6.         <Setter Property="VerticalOptions"

```

```

7.         Value="Center" />
8.     </Style>
9. </Grid.Resources>
10.    <Frame Background="LightBlue"
11.        Grid.ColumnSpan="2">
12.        <VerticalStackLayout Style=
13.            "{StaticResource centerFrame}">
14.            <Label Text="ENTER BILL AMOUNT"/>
15.            <Entry />
16.        </VerticalStackLayout>
    </Frame>

```

As we have applied the style to the **VerticalStackLayout** to center align the child controls, the **Label** and **Entry** controls will be positioned in the center of the layout as shown in [Figure 7.9](#):

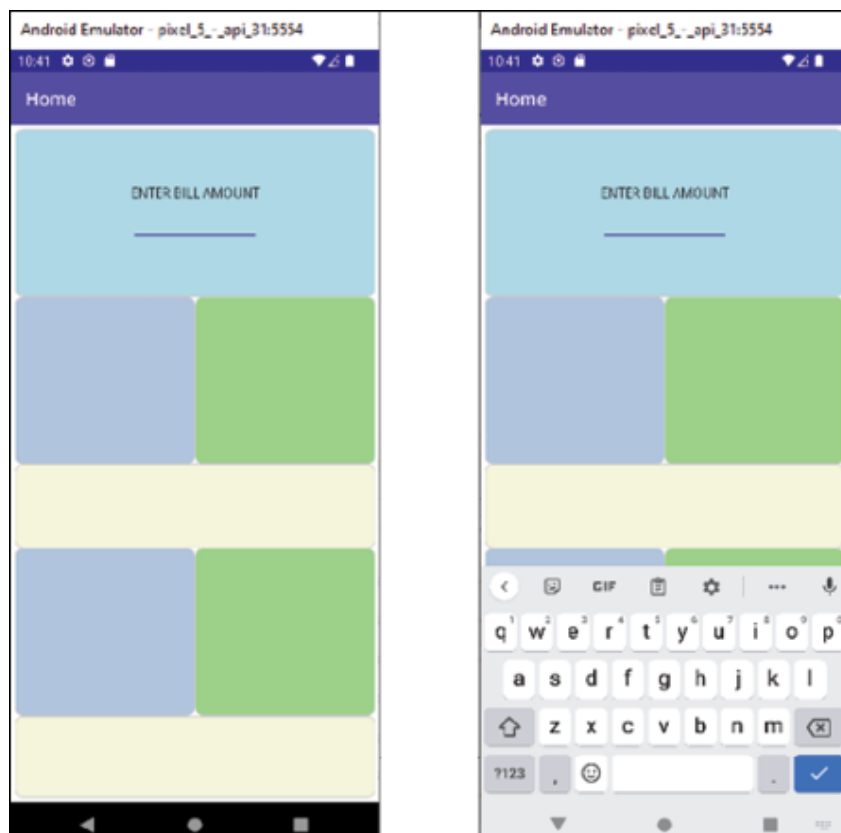


Figure 7.9: Controls for entering Bill Amount

Note: Notice the Entry displays alphanumeric keyboard by default.

8. Let us add the remaining controls for the UI in the same way. In row 2, for the Tip %, add a label to denote the tip selected by the user using a slider control in row 3. The tip amount will be calculated using C# later and it

- will appear under the tip amount.
9. In row 3, place a slider to adjust the tip % mentioned in *Step 9*.
 10. In row 4, we have two sets of controls. In the first column, we will display the number of people who will split the total bill amount (that is, bill + tip amount). The number of people will be set by button controls in row 5. The second column will display the amount that each person has to pay, that is, the split amount.
 11. After completing the above steps, the UI will appear as shown in [Figure 7.10](#):

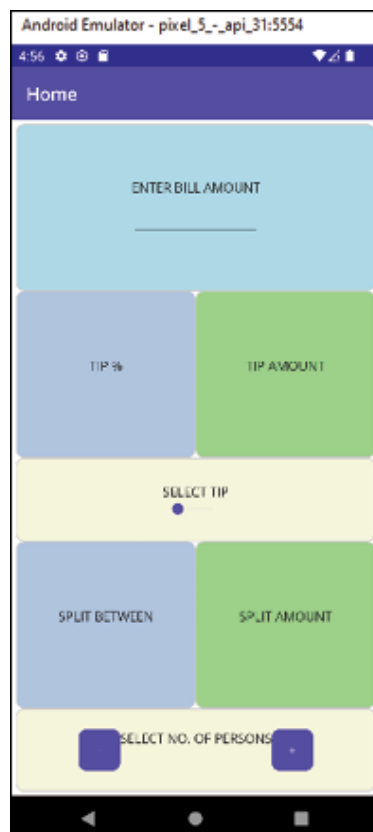


Figure 7.10: Initial state of Data Entry section

Applying graphical design principles in UI design

Principles of graphical design give designers a set of guidelines for how to design visually appealing compositions that create wonderful user experiences. By following basic principles of design like **Contrast, Repetition, Alignment, and Proximity (C.R.A.P.)**, you can create UIs that people love to use. A gist of the C.R.A.P. principles is given in [Figure 7.11](#):

CONTRAST Organize your design, establish hierarchy, emphasizes the focal point, adds visual interest	REPETITION Creates consistency, increases users' learnability, reduces confusion
ALIGNMENT Organizes and groups elements, creates rhythm, brings order to chaos	PROXIMITY Makes elements seem related, brings understanding and meaning.

Figure 7.11: C.R.A.P. principles of design

Right now, our UI looks basic and crude. We will now apply some graphical design principles to make the UI look even better. Follow these steps to achieve it:

1. Firstly, we will introduce some space between the **Frame** controls. This can be done in two ways:
 - a. By declaring **Grid.RowSpacing** and **Grid.ColumnSpacing** properties in the **Grid**.
 - b. By applying margins to the **Frame** controls.
 - c. We will use the latter approach.
2. The margin property of a control represents the distance between the element and its adjacent elements or neighbors. There are three possible ways to assign a margin to the control:
 - a. Margin defined by a single value (for example, **Margin="10"**): The single value is applied to the left, top, right, and bottom sides of the element.
 - b. Margin defined by two values (for example, **Margin="5,10"**): The first value is applied to the left and right of the element and the second value is applied to the top and bottom of the element.
 - c. Margin defined by four distinct values (for example, **Margin="5,10,15,20"**): The values are applied to the left, top, right, and bottom of the element.
3. Using the above understanding, apply appropriate margins to the **Frame** controls so that they are not touching each other, but at the same time, related controls appear together to indicate some kind of association between these controls.
4. Next, let us customize the button controls. We can use the button

properties for **BackgroundColor**, **VerticalOptions**, **HorizontalOptions**, **WidthRequest**, and **FontSize** to customize the **Button**. As the style has to be applied to both the buttons in row 5, we can define this as a style in **Grid**. Resources collection and then using the markup extension syntax we can bind the style property of the button to the customized style that we create.

5. Now, we will customize the labels displaying the numerical values in the UI so that there is an emphasis on the values displayed in the UI. We need to use **FontSize** and **FontAttributes** to our custom style for the **Label** control. Then apply the style to all the Labels which have to display numerical data.

6. After the above changes are done, the XAML markup under **Grid.Resources** would appear as shown in the following code:

```
1. <Grid.Resources>
2.     <Style TargetType="Frame"
3.         x:Key="centerFrame">
4.         <Setter Property="HorizontalOptions"
5.             Value="Center" />
6.         <Setter Property="VerticalOptions"
7.             Value="Center" />
8.     </Style>
9.     <Style TargetType="Button"
10.        x:Key="splitButton">
11.        <Setter Property="BackgroundColor"
12.            Value="Orange" />
13.        <Setter Property="VerticalOptions"
14.            Value="Center" />
15.        <Setter Property="HeightRequest"
16.            Value="50" />
17.        <Setter Property="WidthRequest"
18.            Value="50" />
19.        <Setter Property="FontSize"
20.            Value="Body" />
21.    </Style>
22.    <Style TargetType="Label"
23.        x:Key="displayValue">
24.        <Setter Property="FontSize"
25.            Value="36" />
```

```

26.         <Setter Property="FontAttributes"
27.             Value="Bold" />
28.     </Style>
29. </Grid.Resources>

```

To apply the styles to the **individual controls**, apply the styles using the markup extensions as follows:

```

1. <Label x:Name="lblTip" Style="
    {StaticResource displayValue}" />

```

Apply the style that we created for **Frame** under window. Resources to different Frames. Refer to the following code:

```

1. <Frame Background="LightSteelBlue"
2.     Margin="10,10,5,5"
3.     Grid.Row="1">
4. <VerticalStackLayout Style="
    {StaticResource centerFrame}">

```

Similarly, add the style to the **Button**. Refer to the following code:

```

1. <Button x:Name="btnPlus"
2.     Style="{StaticResource splitButton}"
3.     Text="+"
4.     Clicked="btnPlus_Clicked"></Button>

```

7. With the above XAML markup our UI should look better than before, as shown in [Figure 7.12](#):

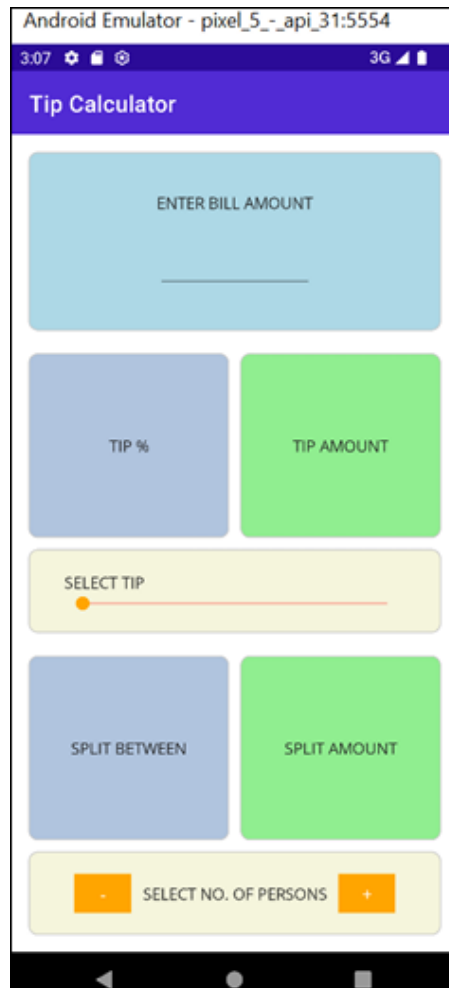


Figure 7.12: Improvised UI

Performing the calculation

Now that the UI is complete, let us turn our attention to the functionality. Follow these steps to define the functionality of the app:

1. In the code-behind file, let us define variables to define the bill amount, tip %, tip amount, number of people who will be splitting the bill and the amount to be paid by each person. Now, define the variables **billAmount**, **tipAmount** and **amountPerPerson** variables as shown in the following code:

```

2.    decimal billAmount = 0.00m;
3.
4.    int tip = 0;
5.    decimal tipAmount = 0.00m;
6.
7.    int split = 1;
8.    decimal amountPerPerson = 0.00m;

```

2. Observe the use of the suffix `m` which is required while assigning values to variables of type `decimal`.
3. Next, let us declare an **UpdateDisplay()** function which will convert the values of the above variables to `String` and assign to the respective labels that we have already created in the UI:

```
1.     private void UpdateDisplay()
2.     {
3.         lblTip.Text = tip.ToString();
4.         lblTipAmount.Text = tipAmount.ToString("F2");
5.         lblSplit.Text = split.ToString();
6.
7.         lblSplitAmount.Text = amountPerPerson.ToString("F2");
8.     }
```

4. We can call this function when the page loads. We will invoke this method in the Page Constructor as shown in the following code:

```
1.     public MainPage()
2.     {
3.         InitializeComponent();
4.         UpdateDisplay();
5.     }
```

5. Whenever the user enters the bill amount in the **Entry** control, it is named as **tbTotal** in this app, we want to calculate the tip amount and the split amount.
6. Let us implement the code in the completed event handler of the entry:

```
1.     private void tbTotal_Completed(object sender, EventArgs e)
2.     {
3.         billAmount = decimal.Parse(tbTotal.Text);
4.         Calculate();
5.         tbTotal.Unfocus();
6.         slrTip.Focus();
7.     }
8.
9.     private void Calculate()
10.    {
11.        tipAmount = billAmount * tip / 100;
```

```

12.         amountPerPerson = (billAmount + tipAmount) / split;
13.         UpdateDisplay();
14.     }

```

7. Notice that the code also includes statements to disable focus on the **Entry** and focus on the **Slider** to adjust the tip %. Whenever the slider value is changed, we want the values to be updated, which is achieved with the following code:

```

1.     private void slrTip_ValueChanged(object sender, ValueChangedEventArgs e)
2.     {
3.         tip = (int)slrTip.Value;
4.         Calculate();
5.     }

```

8. To adjust the number of people who would be sharing the bill, we have two buttons. The event handlers of these buttons can be used to calculate the updated values. Refer to the following code:

```

1.     private void btnMinus_Clicked(object sender, EventArgs e)
2.     {
3.         if (split > 1)
4.             split--;
5.         Calculate();
6.
7.     }
8.
9.     private void btnPlus_Clicked(object sender, EventArgs e)
10.    {
11.        split++;
12.        Calculate();
13.    }

```

9. Run the app after making the above changes and test all the functionalities.
10. Now that our app is complete, we have one more change to do. Notice

that the header in the UI has been displaying the default text **Home**. Edit the **AppShell.xaml** file and replace the **Title** attribute of the **ShellContent** class with the string **Tip Calculator** or your preferred app name. Refer to the following code:

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <Shell
3.     x:Class="TipCalculator.AppShell"
4.
5.     xmlns="http://schemas.microsoft.com/dotnet/2021/maui
6.     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
7.     xmlns:local="clr-namespace:TipCalculator"
8.     Shell.FlyoutBehavior="Disabled">
9.     <ShellContent
10.         Title="Tip Calculator"
11.         ContentTemplate="{DataTemplate local:MainPage}"
12.         Route="MainPage" />
13.
14. </Shell>
```

11. Run the app after making the above change, and you should be able to see the final app view, as shown in the following figure:

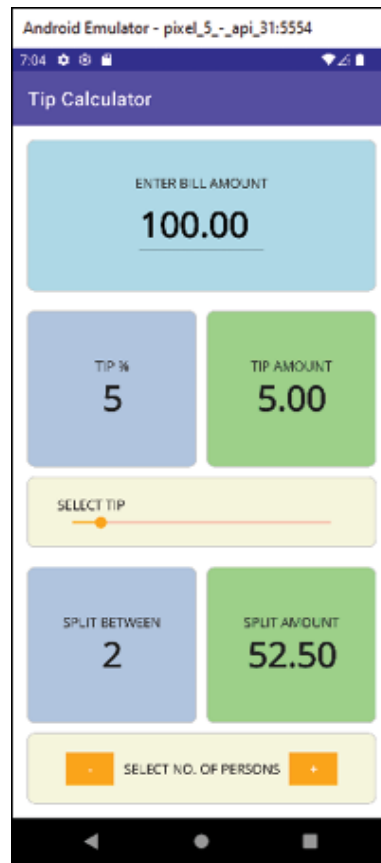


Figure 7.13: Final view of the UI

Conclusion

In this chapter, you have developed your second project using .NET MAUI and will be able to run it on your device as well. If you face any errors, check the source code provided for reference, compare it with your code, and make the necessary corrections. In the next chapter, we will improve the skills further by developing a BMI calculator.

Points to remember

Here are some key takeaways from this chapter:

- Ensure that you choose the right .NET Framework version while creating the project.
- This app demonstrated how you can develop a complex UI using different .NET MAUI controls. We have used a Grid inside the ContentPage and then placed frames inside the Grid which in turn hosted the VerticalStackLayout and HorizontalStackLayout controls. However, you may also develop the UI using alternative arrangement of these controls.

- We discussed on following graphical design principles while laying out the controls. While the C.R.A.P. principles are important to improve the usability and aesthetics of the app, in a real-world app, there are many other design features that are equally important which is beyond scope of this book.
- If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.

CHAPTER 8

Project-3: BMI Calculator

Introduction

A **Body Mass Index (BMI)** calculator is a tool or app used to assess an individual's body weight in relation to their height. It serves as a basic indicator of whether a person's weight falls within a healthy range or deviates from it. BMI is widely used in healthcare, fitness, and wellness programs to provide a simple yet informative snapshot of one's health. BMI is calculated using the formula: $BMI = weight / (height * height)$, where weight is the person's weight in kilograms (kg) and height is the person's height in meters (m). For example, if someone weighs 70 kg., and is 1.70 m tall, their BMI would be calculated as follows:

$$BMI = 70 / (1.7 * 1.7) = 70 / 2.89 = 24.22$$

The resulting value is then interpreted with the predefined BMI categories to determine if the individual is underweight, normal weight, overweight, or obese.

The BMI classifications are as follows:

- under 18.5—This is described as underweight
- between 18.5 and 24.9—This is described as the healthy range

- between 25 and 29.9—This is described as overweight
- between 30 and 39.9—This is described as obesity
- 40 or over—This is described as severe obesity

The app that we will build in this chapter is not an exhaustive BMI calculator catering to all age groups but rather a simplified version aimed at showcasing key concepts of .NET MAUI. It demonstrates the fundamental functionalities of a BMI calculator, offering users an interactive way to calculate their BMI and receive a basic understanding of their weight status.

Structure

In this chapter, we will discuss the following topics:

- Creating the layout
- Creating a data entry section
- Data binding between UI controls
- Data binding to an object
- Binding value converters

Objectives

After reading this chapter, you will be able to:

- Create a UI layout using .NET MAUI controls such as Grid, Frame, VerticalStackLayout, Label, and Slider controls.
- You will also be able to develop the UI in XAML along with understanding how to implement data binding between UI controls.
- This chapter will also help you understand how to implement data binding between UI control and data model.
- By the end of this chapter, you will learn how to implement the functionality to calculate the BMI value in C# and how to use a value converter to determine the weight status.

Creating the layout

Follow these steps to create a layout for the app:

1. Start Visual Studio.
2. In the dialog that appears, select the option: **Create a new project** as shown in *Figure 8.1*:

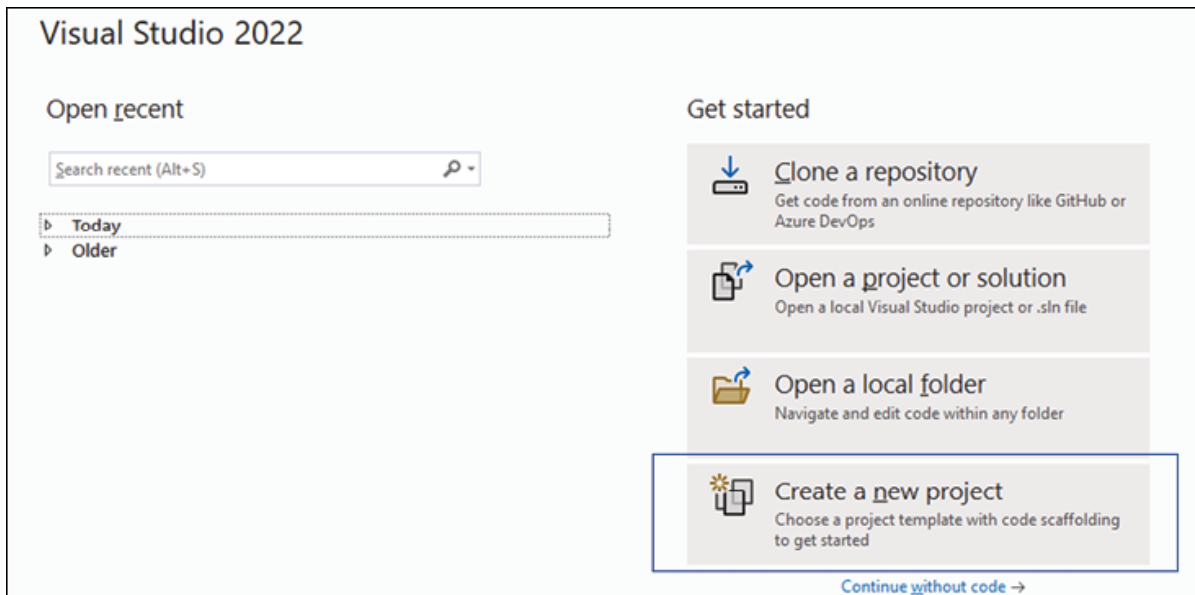


Figure 8.1: *Creating a new project in Visual Studio*

3. Select the programming language as C# and MAUI project to view the MAUI project templates as shown in *Figure 8.2*:

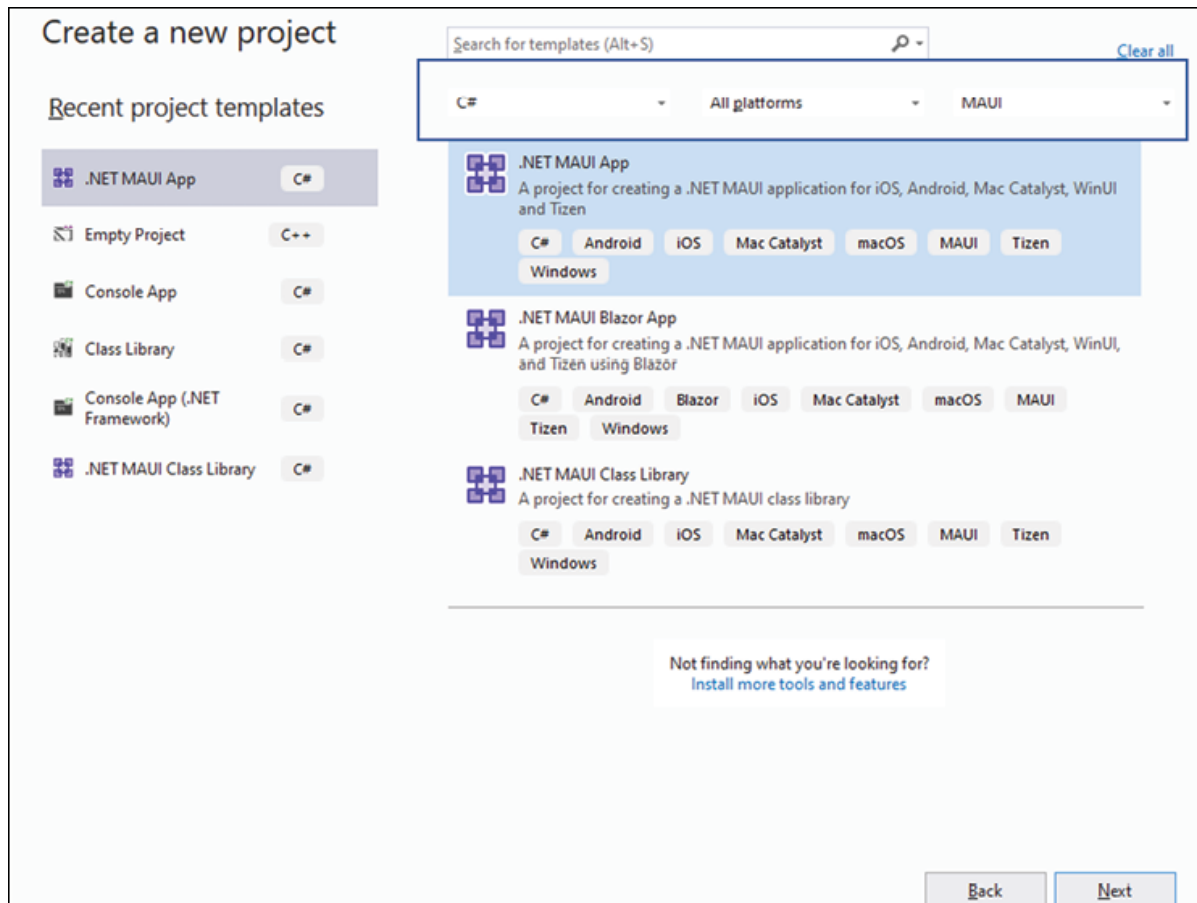


Figure 8.2: Selecting .NET MAUI project template

4. Select .NET MAUI App and click on **Next**.
5. Enter the Project name as **BMICalculator**. Select the preferred destination folder or leave it with the default setting, as shown in [Figure 8.3](#):

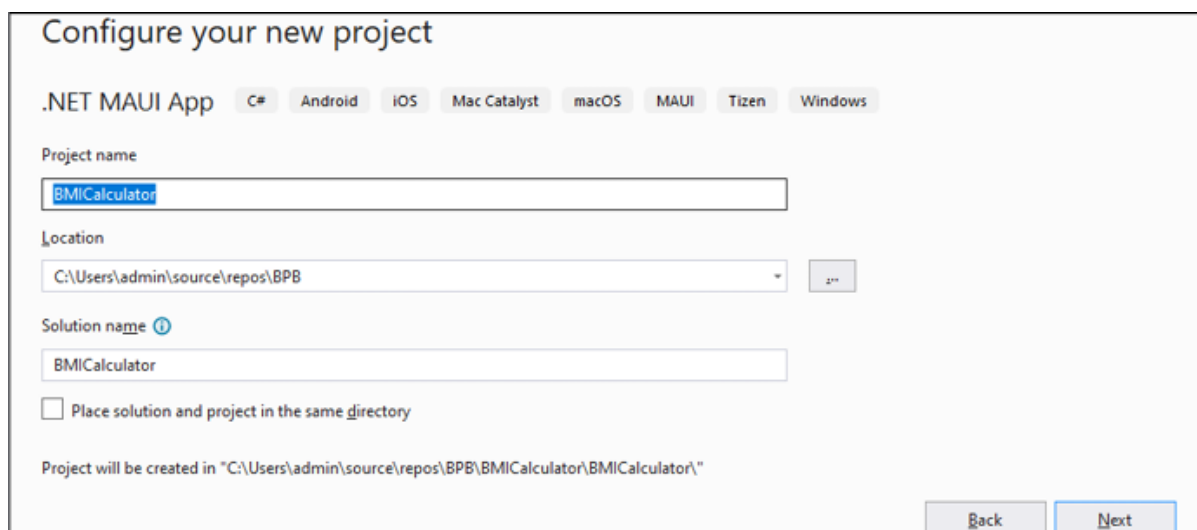


Figure 8.3: Configure your project

6. You can choose the latest **Framework** version which is displayed under **Additional information** and click on **Create**:



Figure 8.4: Selecting the Framework version

7. The MAUI project is created, and within a few minutes, you should be able to see the project in the **Solution Explorer**, as shown in the following figure:

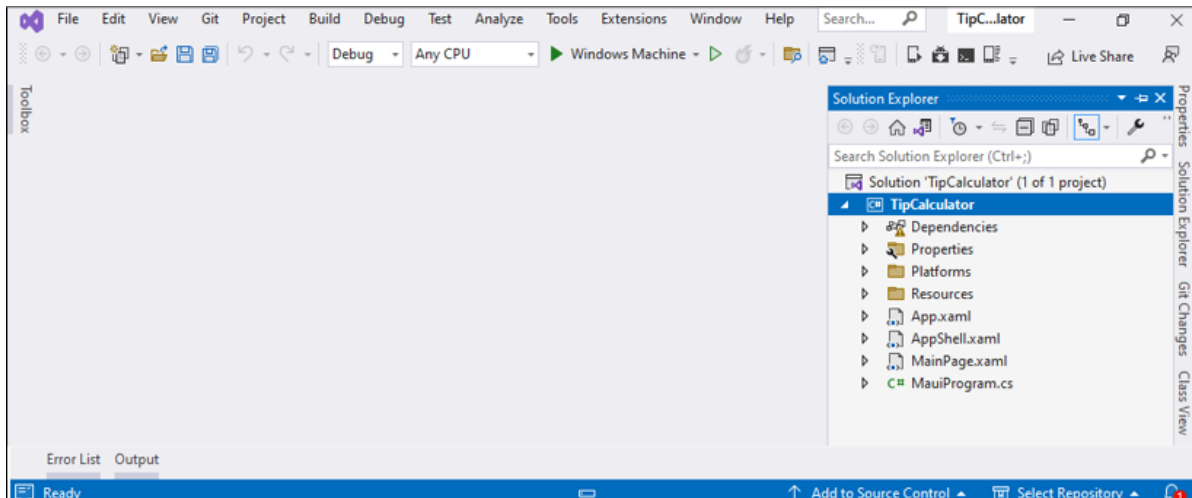


Figure 8.5: Default Solution in Visual Studio

8. Make the following changes to the boilerplate code in **MainPage.xaml**. The **ContentPage** will serve as the parent for all the controls in your UI. Refer to the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/d
   otnet/2021/maui"
3.             xmlns:x="http://schemas.microsoft.com
   /winfx/2009/xaml"
4.             x:Class="BMICalculator.MainPage">
5.
6.
```

```
7. </ContentPage>
```

9. Open the code-behind file **MainPage.xaml.cs** and remove the default project code, as shown in the following code:

```
1. namespace BMICalculator;  
2.  
3. public partial class MainPage : ContentPage  
4. {  
5.     public MainPage()  
6.     {  
7.         InitializeComponent();  
8.     }  
9.  
10. }
```

10. For this app, we will have a **Grid** with multiple rows, which will serve as the container for the data entry controls that we will add next.
11. By default, a **Grid** will be created with one row and column. To create multiple rows and columns, we use the **Grid.RowDefinitions** property of the **Grid**. Each **RowDefinition** declared inside the **Grid.RowDefinitions** creates a row. For our app, we will need three rows.
12. The height of the rows can be defined in three ways:
 - a. **Fixed value**: To assign a fixed size of logical units.
 - b. **Auto**: To take up only as much space that is required for the controls in that specific row.
 - c. **Star**: For proportional sizing to defining sizes relative to each other based on proportions or percentages. This allows us to allocate available space dynamically among the rows based on their specified proportions, which is better for creating flexible and responsive layouts.
13. We will use the proportional sizing approach to define the heights of the rows. As the app will have minimum number of controls that we will place inside the respective rows, we will define equal proportions for the row height. Refer to the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>  
2. <ContentPage xmlns="http://schemas.microsoft.com/d
```

```

otnet/2021/maui"
3.         xmlns:x="http://schemas.microsoft.com
/winx/2009/xaml"
4.         x:Class="BMICalculator.MainPage">
5.
6.     <Grid RowSpacing="10"
7.         Background="LightGray">
8.         <Grid.RowDefinitions>
9.             <RowDefinition Height="*" />
10.            <RowDefinition Height="*" />
11.            <RowDefinition Height="*" />
12.        </Grid.RowDefinitions>
13.
14.    </Grid>
15.</ContentPage>

```

14. Any control can be placed within a **Grid** by using its **Grid.Row** property that represent which row the control will be placed in. The values of rows start with **0**. This means that if there are two rows in the **Grid**, the first row will be represented by the value **Grid.Row=0** and the second row by the value **Grid.Row=1**, and so on.

15. For our app, we will place **Frame** controls inside the rows. **Frame** controls are generally used to create borders around related controls to show that they have something in common. It may be worth mentioning here that a **Border** can also be used in place of a **Frame**.

16. Let us assign gray color for the **Grid** using its **Background** property and assign a white color background for the **Frame** controls by using the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/d
otnet/2021/maui"
3.         xmlns:x="http://schemas.microsoft.com
/winx/2009/xaml"
4.         x:Class="BMICalculator.MainPage">
5.
6.     <Grid RowSpacing="10"
7.         Background="LightGray">

```

```

8.      <Grid.RowDefinitions>
9.          <RowDefinition Height="*" />
10.         <RowDefinition Height="*" />
11.         <RowDefinition Height="*" />
12.     </Grid.RowDefinitions>
13.
14.     <Frame Background="White">
15.     </Frame>
16.
17.     <Frame Background="White"
18.         Grid.Row="1">
19.     </Frame>
20.
21.     <Frame Background="White"
22.         Grid.Row="2">
23.     </Frame>
24. </Grid>
25. </ContentPage>

```

17. At this stage, the UI should look like the illustration in *Figure. 8.6*:

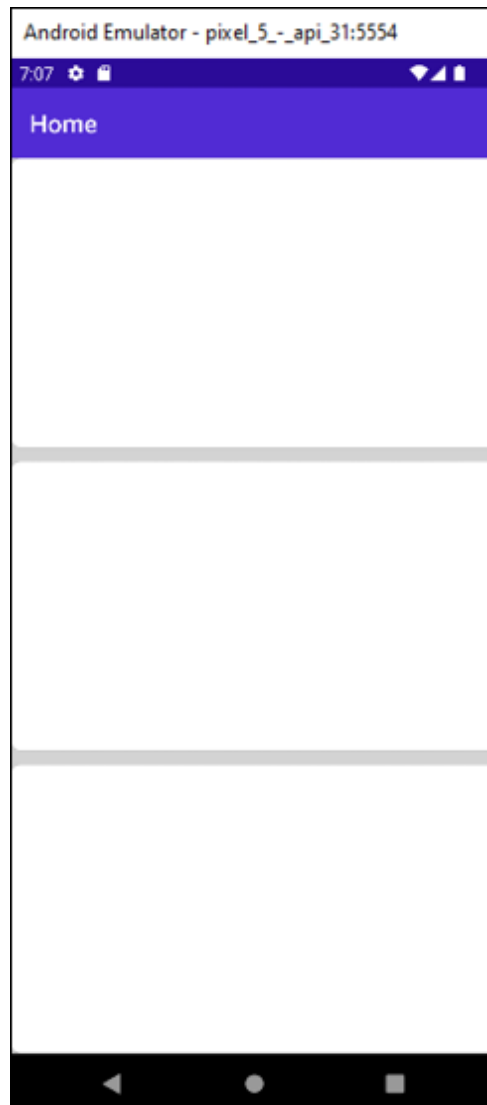


Figure 8.6: Layout of the app

Creating a data entry section

Follow these steps to insert the necessary controls to develop the UI that forms the data entry section:

1. In the previous section, we added Frames inside the **Grid**. As the **Frame** can take only one child, we will use a layout container like **VerticalStackLayout** as the child of the **Frame**. We will then place the other controls of the UI inside it. Refer to the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui">
```

```

3.         xmlns:x="http://schemas.microsoft.com
/winfx/2009/xaml"
4.         x:Class="BMICalculator.MainPage">
5.
6.     <Grid RowSpacing="10"
7.         Background="LightGray">
8.         <Grid.RowDefinitions>
9.             <RowDefinition Height="*" />
10.            <RowDefinition Height="*" />
11.            <RowDefinition Height="*" />
12.        </Grid.RowDefinitions>
13.
14.        <Frame Background="White">
15.            <VerticalStackLayout Spacing="20">
16.
17.                </VerticalStackLayout>
18.            </Frame>
19.            <Frame Background="White"
20.                Grid.Row="1">
21.                <VerticalStackLayout Spacing="20">
22.
23.                    </VerticalStackLayout>
24.                </Frame>
25.                <Frame Background="White"
26.                    Grid.Row="2">
27.                    <VerticalStackLayout Spacing="20">
28.
29.                        </VerticalStackLayout>
30.                    </Frame>
31.                </Grid>
32.    </ContentPage>

```

2. To enter the height and weight, we need to provide some controls. In the row 1 frame, let us add the following controls:

- A label to provide the caption to enter **Height**.

- Another label to display the value that will be set by a Slider control, using data binding.
- A Slider control with minimum and maximum values set to 40 and 200 respectively, which represents the height of the person in cm.

Refer to the following code:

```

1. <Frame Background="White">
2.     <VerticalStackLayout Spacing="20">
3.         <Label Text="Height (in cm.)" />
4.         <Label Text="130.0" />
5.         <Slider x:Name="slrHeight"
6.             Minimum="40"
7.             Maximum="200" />
8.     </VerticalStackLayout>
9. </Frame>

```

3. We can add a similar set of UI controls in Row 2 for selecting the weight in kg:

```

1. <Frame Background="White"
2.     Grid.Row="1">
3.     <VerticalStackLayout Spacing="20">
4.         <Label Text="Weight (in kg.)" />
5.         <Label Text="70.5"/>
6.         <Slider x:Name="slrWeight"
7.             Minimum="40"
8.             Maximum="200" />
9.     </VerticalStackLayout>
10. </Frame>

```

4. To display the calculated BMI value and the weight status, let us insert **Label** controls in Row 3 as shown in the following code:

```

1. <Frame Background="White"
2.     Grid.Row="2">
3.     <VerticalStackLayout Spacing="20">
4.         <Label />

```

5. `<Label />`
 6. `<Label />`
 7. `</VerticalStackLayout>`
 8. `</Frame>`
5. At this stage, the app should resemble the screenshot as shown in the following figure:

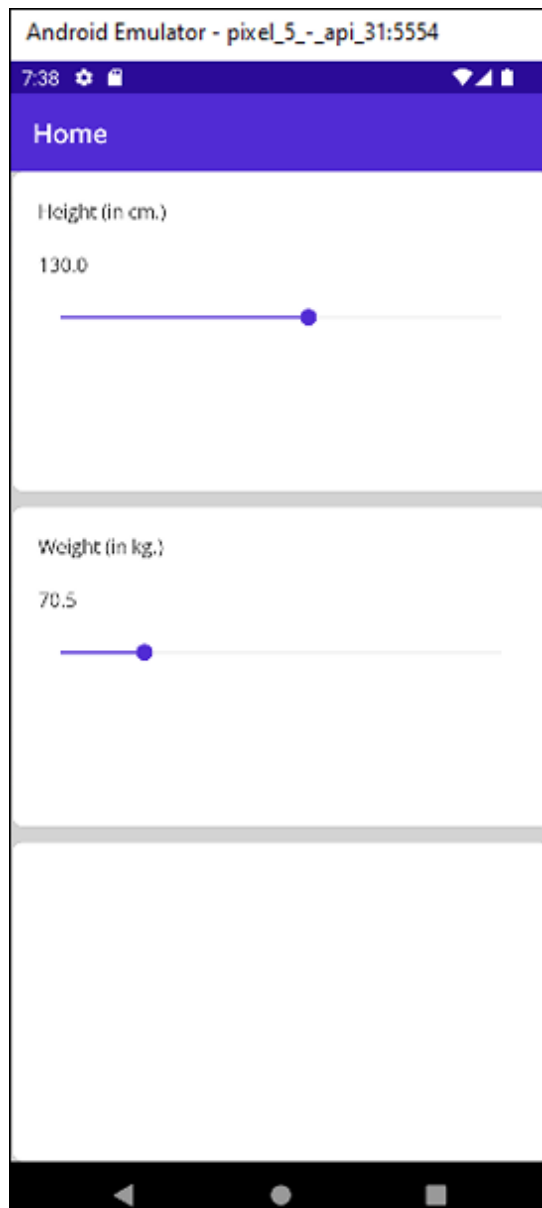


Figure 8.7: The UI without any styles applied to the labels

6. In [Chapter 7](#), *Project-2: Tip Calculator*, we learned how to create named styles and assign it to UI controls. In this simple app, we will use the same style for all the Labels so we can create a default style for the Label controls to display the Text using the **Title** font size.

Refer to the following code:

```
1. <Grid.Resources>
2.     <Style TargetType="Label">
3.         <Setter Property="FontSize"
4.             Value="Title" />
5.     </Style>
6. </Grid.Resources>
```

The data entry section is now complete. At this stage, your app should resemble the following screenshot:

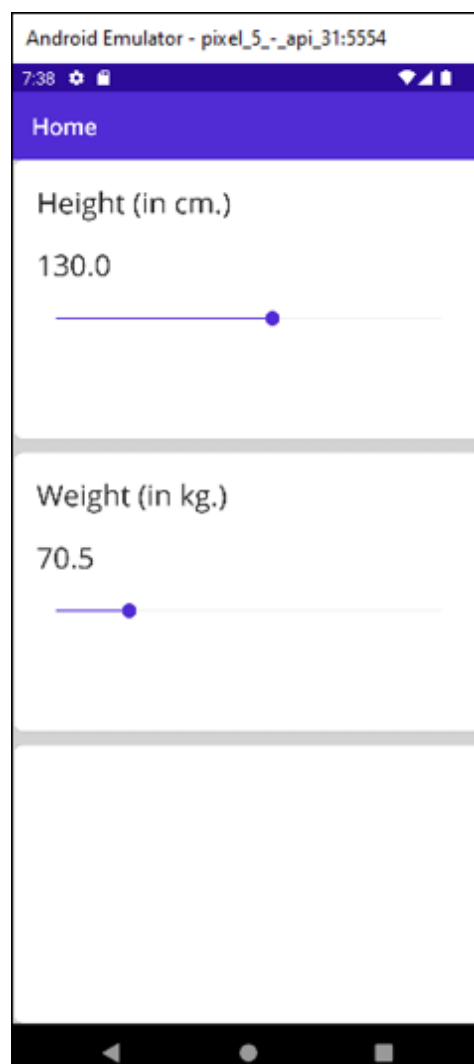


Figure 8.8: The UI after applying a default style to all labels

Next, we will implement data binding in our app.

Data binding between UI controls

Data binding is a technique of linking the properties of two objects so that changes in the property of one object are automatically reflected in the other object's property. One of the two objects involved in data binding is almost always a UI control, and the other object could be either:

- Another UI control
- An object in the code file

In this app, we will implement both the above types of data binding to help you understand them in an easy way. Let us now look at how UI controls—a Label and a Slider—can seamlessly interact with one another through data binding.

1. Our first requirement is to display the value of the Slider in the Label whenever it is adjusted. If data binding is not available, we have to use the Slider's **ValueChanged** event and bind an event handler in which you can access the Slider's **Value** property to the **Text** property of the **Label**. With data binding, this task is automated so there is no need for you to implement the event handler and the code for the same.
2. Data binding can be done either in XAML or using C#, but the XAML approach is easier.
3. The data binding is set on the target object, which in our example is the **Label**. Two XAML markup extensions are used to define the data binding:
 - The **x:Reference** markup extension is used to reference the source object (e.g. **slrHeight**)
 - The **Binding** markup extension links the Label's **Text** property to the Slider's **Value** property.

To bind the data, we use the following expression:

1. `<Label BindingContext="{x:Reference slrHeight}"`
2. `Text="{Binding Path=Value}" />`

4. Additionally, the value can be formatted by providing a placeholder to the **StringFormat** property of the **Binding** markup extension. Ensure to provide single quote characters around the formatting string which will help the XAML parser avoid conflict with the curly braces of the **Binding** markup extension.
5. In order to display the value with one decimal place we use the

placeholder **'{0:F1}'** to set the **StringFormat**:

```
1. <Label BindingContext="{x:Reference slrHeight}"
2.     Text="
    {Binding Path=Value, StringFormat='{0:F1}}'" />
```

6. Repeat the above steps for the section representing **Weight** (in kgs.):

```
1. <Frame Background="White"
2.     Grid.Row="1">
3.     <VerticalStackLayout Spacing="20">
4.         <Label Text="Weight (in kg.)" />
5.         <Label BindingContext="
6.             {x:Reference slrWeight}"
7.             Text="
8.                 {Binding Path=Value, StringFormat='{0:F1}}'" />
9.         <Slider x:Name="slrWeight"
10.             Minimum="40"
11.             Maximum="200"
12.             ValueChanged="slr_ValueChanged" />
13.     </VerticalStackLayout>
14. </Frame>
```

7. Now, run the app and check if the value of the **Slider** controls are displayed in the corresponding **Label** controls when adjusted.

Data binding to an object

In the previous section, we implemented data binding between the UI controls. When the slider value is changed, we can use the Slider's **ValueChanged** event to calculate the BMI in the code-behind file. To update the UI whenever the value in the code-behind file changes, we can implement a class that will be known as a model (or ViewModel if MVVM architectural pattern is used). The model uses a way to let the system know when a property's value changes. This is done through the **INotifyPropertyChanged** interface, which has an event named **PropertyChanged**. When a class follows this interface and one of its properties changes, the event is triggered. If the property does not change, the event will not be triggered. In our solution, the BMI class has a property called **BMIValue**. When this value changes, it triggers the **PropertyChanged** event. Refer to the following code:

```

1. using System;
2. using System.Collections.Generic;
3. using System.ComponentModel;
4. using System.Linq;
5. using System.Runtime.CompilerServices;
6. using System.Text;
7. using System.Threading.Tasks;
8.
9. namespace BMICalculator
10. {
11.     public class BMI : INotifyPropertyChanged
12.     {
13.         private double bmivalue;
14.
15.         public double BMIValue
16.         {
17.             get { return bmivalue; }
18.             set
19.             {
20.                 if(bmivalue != value)
21.                 {
22.                     bmivalue = value;
23.                     NotifyPropertyChanged();
24.                 }
25.             }
26.         }
27.
28.         public event PropertyChangedEventHandler Property
29.         ertyChanged;
30.
31.         private void NotifyPropertyChanged([CallerMemberName]
32.
33.         String propertyName = "")
34.         {
35.             PropertyChanged?.Invoke(this, new

```



```

        PropertyChangedEventArgs(propertyName));
33.     }
34. }
35. }

```

In the code-behind file, we create an instance of the model, that is, **BMI** class. Whenever the Slider value is changed, the BMI is calculated, and the **BMIValue** property is updated, which causes the **Label** control in Grid's Row 3 to reflect the updated value. Refer to the following code:

```

1. namespace BMICalculator;
2.
3. public partial class MainPage : ContentPage
4. {
5.     BMI MyBMI;
6.
7.     public MainPage()
8.     {
9.         InitializeComponent();
10.        MyBMI = new BMI { BMIValue = 0 };
11.        BindingContext = MyBMI;
12.    }
13.
14.    private void slr_ValueChanged(object sender, Value
        eChangedEventArgs e)
15.    {
16.        CalculateBMI();
17.    }
18.
19.    private void CalculateBMI()
20.    {
21.        Double height = slrHeight.Value / 100;
22.        Double weight = slrWeight.Value;
23.
24.        MyBMI.BMIValue = weight / (height * height);
25.    }
26. }

```

Binding value converters

Data binding transfers data from a source property to a target property in one-way binding mode and two-way binding in the reverse direction. When the source and target properties are of the same type, or when the source type can be converted to the target type through an implicit conversion, the data transfer is straightforward. When that is not the case, a **ValueConverter** must be implemented to take place for explicit type conversion.

In our solution, we have calculated the BMI value in the code-behind file. Follow these steps to display the BMI category based on the BMI value that we have calculated:

1. We want to display the weight status also in the UI. While it is easy to create another property of String type in the BMI class and assign the weight status to this property to display it in the UI, we will use this opportunity to demonstrate the concept of a **ValueConverter**.
2. The source in this case is the **BMIValue**, which is of type **Double**, and the target property is a string, which displays one of the four possible strings to indicate the weight status. To achieve this, we implement a **ValueToStringConverter** class that implements a **IValueConverter** interface. Refer to the following code:

```
1. using System;
2. using System.Collections.Generic;
3. using System.Globalization;
4. using System.Linq;
5. using System.Text;
6. using System.Threading.Tasks;
7.
8. namespace BMICalculator
9. {
10.     public class ValueToStringConverter : IValueCo
        nverter
11.     {
12.         public object Convert(object value, Type t
            argetType,
            object parameter, CultureInfo culture)
13.         {
```

```

14.         throw new NotImplementedException();
15.     }
16.
17.     public object ConvertBack(object value,
    Type targetType, object parameter, CultureInfo cul-
    ture)
18.     {
19.         throw new NotImplementedException();
20.     }
21. }
22. }

```

3. You can then create an instance of this class using XAML markup, as shown in the source code. Notice the use of an additional attribute **xmlns:local** in XAML which is required when you are instantiating an object in XAML. You can then set this instance to the **Converter** property of the **Binding** markup extension. Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/d
    otnet/2021/maui"
3.     xmlns:x="http://schemas.microsoft.com
    /winfx/2009/xaml"
4.     xmlns:local="clr-
    namespace:BMICalculator"
5.     x:Class="BMICalculator.MainPage">
6.
7.     <ContentPage.Resources>
8.         <local:ValueToStringConverter x:Key="valTo
    String" />
9.     </ContentPage.Resources>

1. <Frame Background="White"
2.     Grid.Row="2">
3.     <VerticalStackLayout Spacing="20">
4.         <Label Text="Your BMI" />
5.         <Label Text="{Binding Path=BMIValue,

```

```

        StringFormat='{0:F1}'}" />
6.         <Label Text="{Binding Path=BMIValue,
           Converter={StaticResource valToString}}" />
7.     </VerticalStackLayout>
8. </Frame>

```

4. The **Convert** method is called when data moves from the source property to the target property in the binding. The argument named value is the object or value from the data binding source. In our example, this is the **BMIValue**. The method can now return a value of the type of the data binding target. The following code will return the weight status as a String, representing the BMI category depending on the value of the calculated BMI. Refer to the following code:

```

1. using System;
2. using System.Collections.Generic;
3. using System.Globalization;
4. using System.Linq;
5. using System.Text;
6. using System.Threading.Tasks;
7.
8. namespace BMICalculator
9. {
10.     public class ValueToStringConverter : IValueCo
        nverter
11.     {
12.         private const Double LIM_NORMALWEIGHT = 18
            .5F;
13.         private const Double LIM_OVERWEIGHT = 25.0
            F;
14.         private const Double LIM_OBESEWEIGHT = 30.
            0F;
15.
16.
17.         public object Convert(object value, Type t
            argetType,
            object parameter, CultureInfo culture)

```

```

18.         {
19.             Double bmi = (Double)value;
20.             String result = null;
21.             /*
22.              *
23.              Underweight: BMI is less than 18.5
24.              Normal weight: BMI is 18.5 to 24.9
25.              Overweight: BMI is 25 to 29.9
26.              Obese: BMI is 30 or more
27.              *
28.              * */
29.
30.             if(bmi < LIM_NORMALWEIGHT) { result =
"Under Weight"; }
31.
32.             if (bmi >= LIM_NORMALWEIGHT && bmi <
LIM_OVERWEIGHT) { return "Normal Weight"; }
33.
34.             if (bmi >= LIM_OVERWEIGHT && bmi <
LIM_OBESEWEIGHT) { return "Over Weight"; }
35.
36.             if (bmi >= LIM_OBESEWEIGHT)
{ return "Obese Weight"; }
37.
38.             return result;
39.         }
40.
41.         public object ConvertBack(object value,
Type targetType, object parameter, CultureInfo cul
ture)
42.         {
43.             throw new NotImplementedException();
44.         }
45.     }
46. }

```

5. The **ConvertBack** method is called when the data moves from target to source. This happens in **TwoWay** or **OneWayToSource** bindings for the opposite conversion. In our case, this does not require implementation.
6. After implementing the above, you can run the app and check if the weight status is getting reflected in the UI when the BMI value changes while the **Slider** controls are manipulated.

Refer to the following figure:

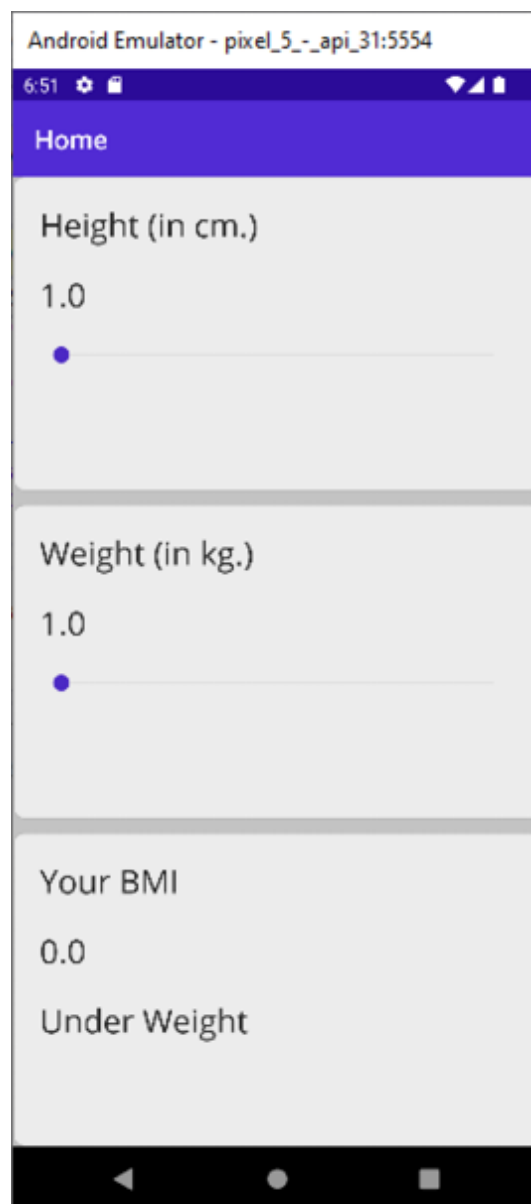


Figure 8.9: After implementing data binding

7. Finally, ensure updating the **Title** in the **AppShell.xaml** file with the name of the app, for example, **BMI Calculator**, by using the

following code:

```
1. <Shell
2.     x:Class="BMICalculator.AppShell"
3.     xmlns="http://schemas.microsoft.com/dotnet/202
4.     xmlns:x="http://schemas.microsoft.com/winfx/20
5.     xmlns:local="clr-namespace:BMICalculator"
6.     Shell.FlyoutBehavior="Disabled">
7.
8.     <ShellContent
9.         Title="BMI Calculator"
10.        ContentTemplate="
11.        {DataTemplate local:MainPage}"
12.        Route="MainPage" />
13. </Shell>
```

The following figure is an illustration of the final design of the BMI calculator:

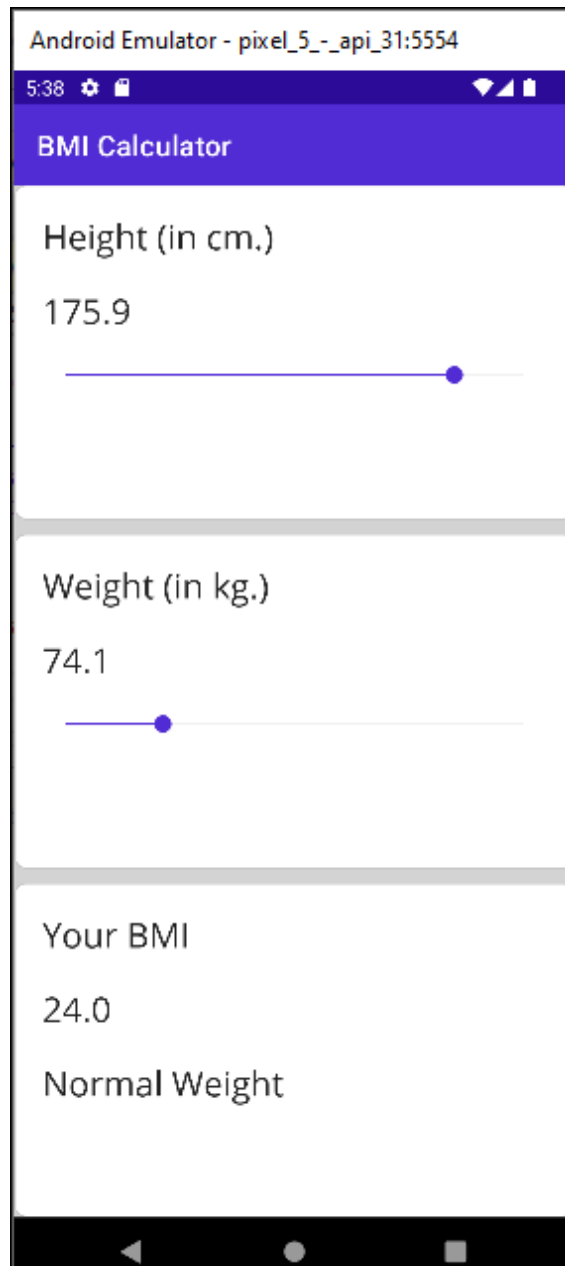


Figure 8.10: Final design of the BMI Calculator

Conclusion

In this chapter, you have developed your third project using .NET MAUI. You will also be able to run it on your device. In case you face any errors, check the source code provided for reference, compare it with your code, and make the necessary corrections.

In the next chapter, we will work on improving our skills further by developing a Unit Converter.

Points to remember

Here are some key takeaways from this chapter:

- Ensure that you choose the right .NET Framework version while creating the project.
- This app demonstrated how you can implement data binding in different scenarios.
- Data binding can be implemented either using XAML or C#.
- The bindingcontext property is assigned to an x:Reference markup extension to refer to another control. The source property is specified with the Path property. While binding to objects, Binding Context can be set in C#.
- It is important to implement INotifyPropertyChanged for the data binding mechanism to be notified of updates when the model is changed.
- Use StringFormat and ValueConverter where necessary to change the format or type of data displayed by the target control.
- If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Project-4: Unit Converter

Introduction

A Unit Converter is an invaluable tool that simplifies the conversion of measurements from one unit to another. Doing these conversions manually can become complex, even using a calculator. Unit converters are especially useful for students and a wide range of professionals in various fields who must perform unit conversions as a part of their day-to-day work.

In this chapter, we will develop a Unit Converter app that will help users convert values swiftly and accurately among diverse units of measurement related to length, weight, volume, area, temperature, and time. The focus will be on understanding how to use .NET MAUI to develop such an app and not on applying best practices using design patterns. The application structure is intentionally kept simple so that a beginner can also follow it easily. Advanced readers might want to implement it using the **Model-View-ViewModel (MVVM)** pattern, which provides the benefit of separating the concerns, making the code more maintainable, and promoting testability.

Structure

In this chapter, we will discuss the following topics:

- Creating the main menu layout
- Creating a navigational system
- Creating the data entry section
- Performing the conversions

Objectives

After reading this chapter, you will be able to:

- Create a navigation system to allow users to navigate between the main menu and the detail pages, both forward and backward.
- You will also be able to develop the UI for the app pages in XAML and implement the logic to convert values from one unit to another in C#.

Creating the main menu layout

Follow these steps to create a new app for the Unit Converter:

1. Open Visual Studio.
2. In the dialog that appears, select the option: **Create a new project**, as shown in *Figure 9.1*:

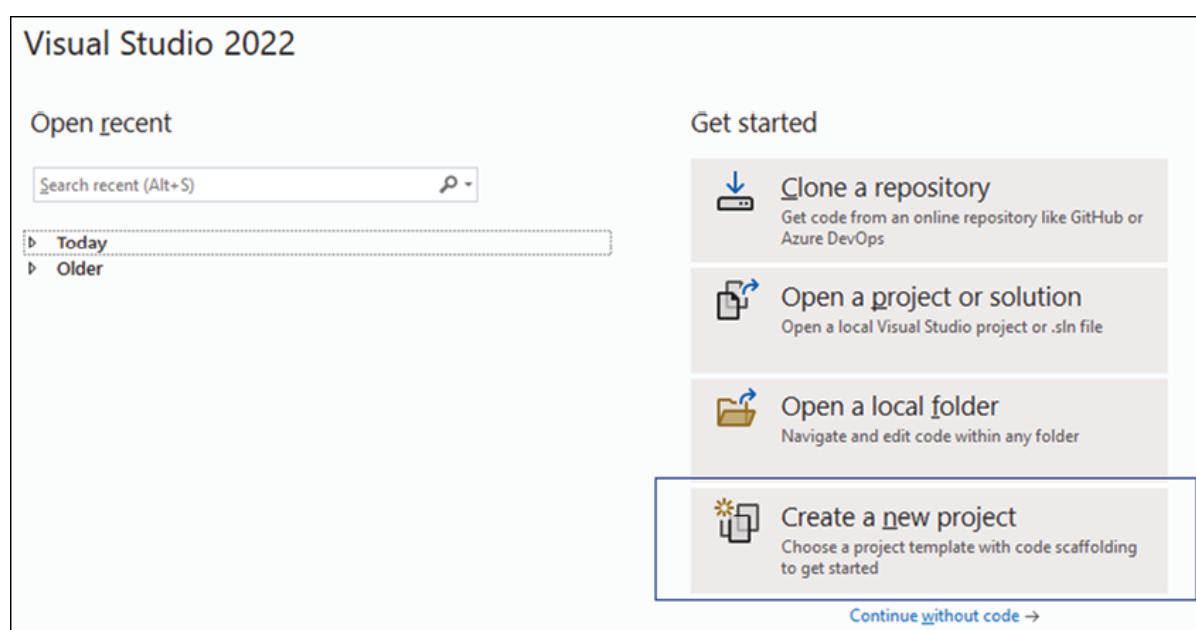


Figure 9.1: Creating a new project in Visual Studio

3. Select the programming language as C# and the Project type as MAUI project to view the MAUI project templates as shown in *Figure 9.2*:

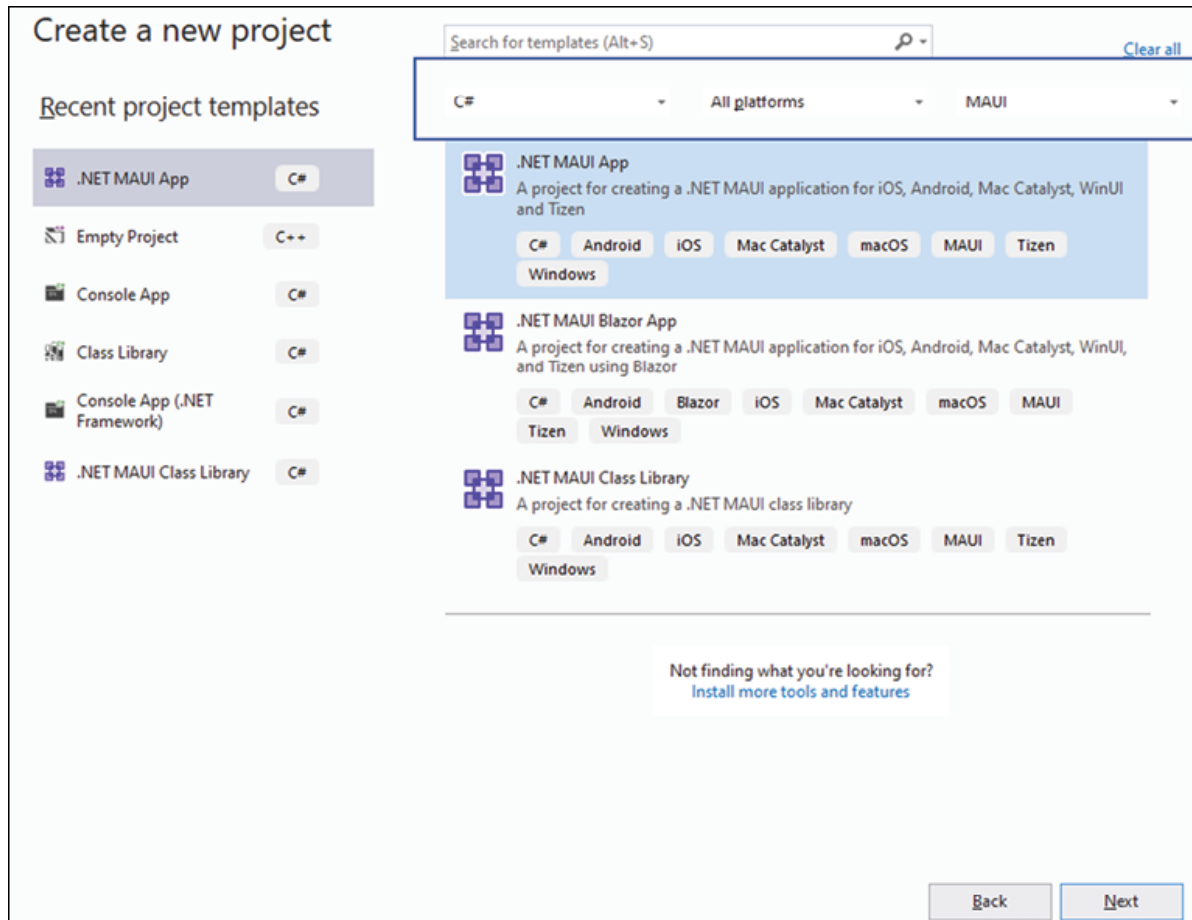


Figure 9.2: Selecting .NET MAUI project template

4. Select **.NET MAUI App** and click on **Next**.
5. Enter the Project name as **UnitConverter**. Select the preferred destination folder or leave it with the default setting, as shown in *Figure 9.3*:

Configure your new project

.NET MAUI App C# Android iOS Mac Catalyst macOS MAUI Tizen Windows

Project name
UnitConverter

Location
C:\Users\admin\source\repos\BPB

Solution name ⓘ
UnitConverter

☐ Place solution and project in the same directory

Project will be created in "C:\Users\admin\source\repos\BPB\UnitConverter\UnitConverter"

Back Next

Figure 9.3: Configure your new project

- You can choose the latest framework version which is displayed under **Additional information** and click on **Create**, as shown in [Figure 9.4](#):

Additional information

.NET MAUI App C# Android iOS Mac Catalyst macOS MAUI Tizen Windows

Framework ⓘ
.NET 7.0 (Standard Term Support)

Back Create

Figure 9.4: Selecting the framework version

- The MAUI project is created, and within a few minutes, you should be able to see the project in the **Solution Explorer**, as shown in [Figure 9.5](#):

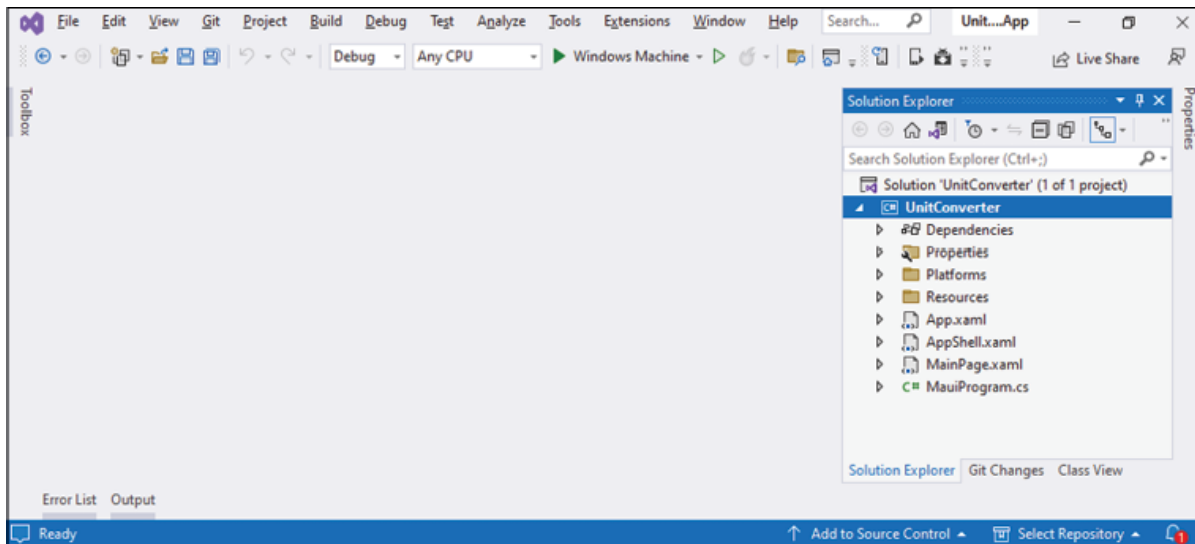


Figure 9.5: Default Solution in Visual Studio

8. Modify the default markup from the **MainPage.xaml** file, as shown in *Figure 9.6*:

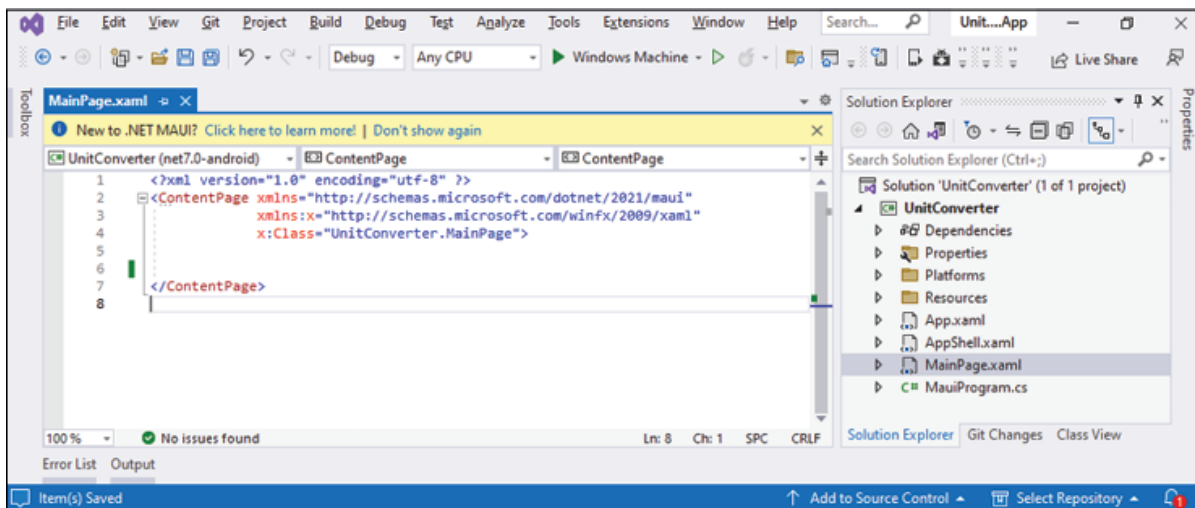


Figure 9.6: MainPage.xaml after modification

9. Modify the default C# code from the code-behind file as well, as shown in *Figure 9.7*:

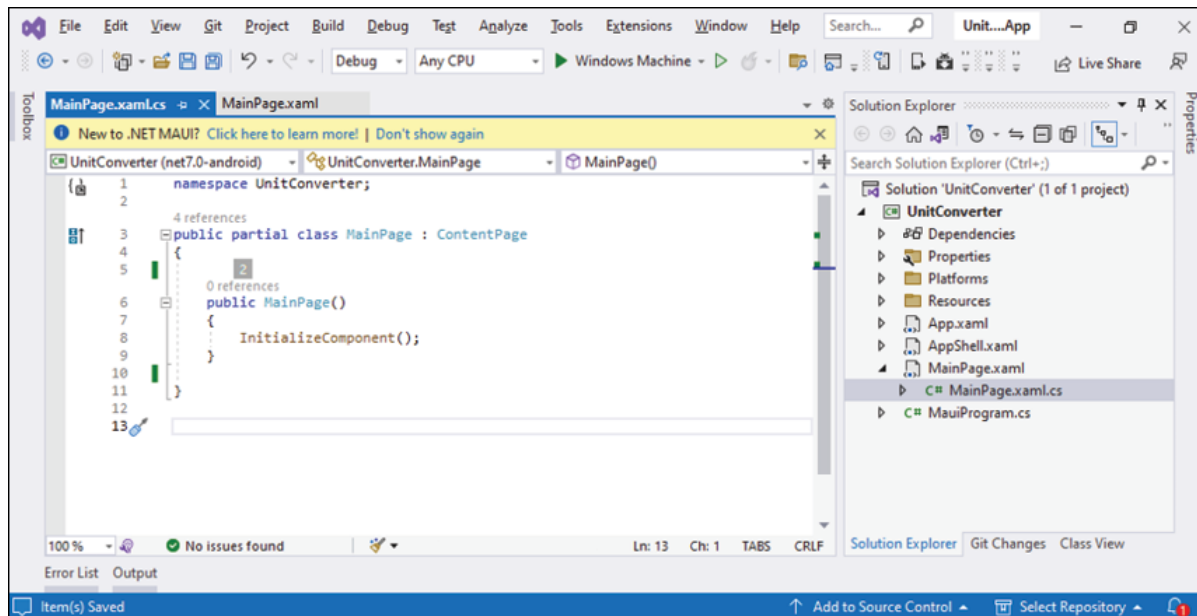


Figure 9.7: MainPage.xaml.cs after modification

10. In our app, we will implement a menu in **MainPage.xaml**. The menu will consist of multiple buttons that will help users navigate to different pages for unit conversion.
11. While the main menu can be implemented in various ways—using a Grid, a WrapPanel, etc., we will use a Grid.
12. Let us define a grid with three rows and two columns, as shown in the below code snippet. Since we have discussed Grid in detail in [Chapter 7, Project-2: Tip Calculator](#), we will not repeat the explanation here. Refer to the following code:

```

1. <Grid ColumnSpacing="10" RowSpacing="10" Padding="
   10">
2.     <Grid.RowDefinitions>
3.         <RowDefinition Height="180" />
4.         <RowDefinition Height="180" />
5.         <RowDefinition Height="180" />
6.     </Grid.RowDefinitions>
7.     <Grid.ColumnDefinitions>
8.         <ColumnDefinition Width="*" />
9.         <ColumnDefinition Width="*" />
10.    </Grid.ColumnDefinitions>
11. </Grid>

```

13. Place the **Button** controls in each cell using the **Grid.Row** and

Grid.Column properties and assign the **Text** property of the **Button** to display the different units of measurement that we will implement in this project. Refer to the following code:

```
1. <Button x:Name="lengthButton"
2.     Text="Length"
3.     Clicked="lengthButton_Clicked"
4.     Grid.Row="0"
5.     Grid.Column="0" />
6.
7. <Button x:Name="weightButton"
8.     Text="Weight"
9.     Clicked="weightButton_Clicked"
10.    Grid.Row="0"
11.    Grid.Column="1" />
12.
13. <Button x:Name="volumeButton"
14.     Text="Volume"
15.     Clicked="volumeButton_Clicked"
16.     Grid.Row="1"
17.     Grid.Column="0" />
18.
19. <Button x:Name="tempButton"
20.     Text="Temperature"
21.     Clicked="tempButton_Clicked"
22.     Grid.Row="1"
23.     Grid.Column="1" />
24.
25. <Button x:Name="areaButton"
26.     Text="Area"
27.     Clicked="areaButton_Clicked"
28.     Grid.Row="2"
29.     Grid.Column="0" />
30.
31. <Button x:Name="timeButton"
32.     Text="Time"
33.     Clicked="timeButton_Clicked"
```



```
34.         Grid.Row="2"
35.         Grid.Column="1" />
```

14. Optionally you can provide different background colors for the buttons in the menu screen using the **BackgroundColor** attribute. Refer to the following code:

```
1. <Button x:Name="lengthButton"
2.         Text="Length"
3.         Clicked="lengthButton_Clicked"
4.         BackgroundColor="LightSeaGreen"
5.         Grid.Row="0"
6.         Grid.Column="0" />
7.
8. <Button x:Name="weightButton"
9.         Text="Weight"
10.        Clicked="weightButton_Clicked"
11.        BackgroundColor="LightCoral"
12.        Grid.Row="0"
13.        Grid.Column="1" />
14.
15. <Button x:Name="volumeButton"
16.        Text="Volume"
17.        Clicked="volumeButton_Clicked"
18.        BackgroundColor="MediumSlateBlue"
19.        Grid.Row="1"
20.        Grid.Column="0" />
21.
22. <Button x:Name="tempButton"
23.        Text="Temperature"
24.        Clicked="tempButton_Clicked"
25.        BackgroundColor="LightSlateGray"
26.        Grid.Row="1"
27.        Grid.Column="1" />
28.
29. <Button x:Name="areaButton"
30.        Text="Area"
31.        Clicked="areaButton_Clicked"
```

```
32.         BackgroundColor="LightSalmon"
33.         Grid.Row="2"
34.         Grid.Column="0" />
35.
36. <Button x:Name="timeButton"
37.         Text="Time"
38.         Clicked="timeButton_Clicked"
39.         BackgroundColor="SkyBlue"
40.         Grid.Row="2"
41.         Grid.Column="1" />
```

15. Remember to change the **Title** attribute of the page as appropriate. In our example, we have renamed it to **Menu**, as shown in the following figure:



Figure 9.8: The completed menu

16. We will implement the **Button_Clicked** event handlers in a later step.

Creating the navigational system

The apps that we developed in Projects 1- 3 were single-page applications and did not present any challenges of navigating between pages. .NET MAUI offers multiple options for developing multi-page apps; **NavigationPage** is one of them. The **NavigationPage** simplifies the

creation of a navigational system. However, we need to define a root page first. Starting from the root page, you can navigate to any content page and then return to the root page effortlessly.

We will set the **MainPage.xaml** created in the previous section, as the root page of our app. Follow these steps to create a navigational system that lets you navigate from the root page to a content page by clicking a button and returning to the root page as well:

1. In the **App.xaml.cs** file, create an instance of the **NavigationPage** and pass an instance of the **MainPage** as an argument to its constructor.
2. This instance of the **NavigationPage** is then assigned to the **MainPage** property of the **App** class, as shown in the following snippet :

```
1. MainPage = new NavigationPage(new MainPage());
```

3. Now, let us add a content page for the **Length** category where we will perform the conversion between different units of measurement related to Length. To do this, right-click on the Project Name in the **Solution Explorer** and select **Add | New Item |** as shown in [Figure 9.9](#):

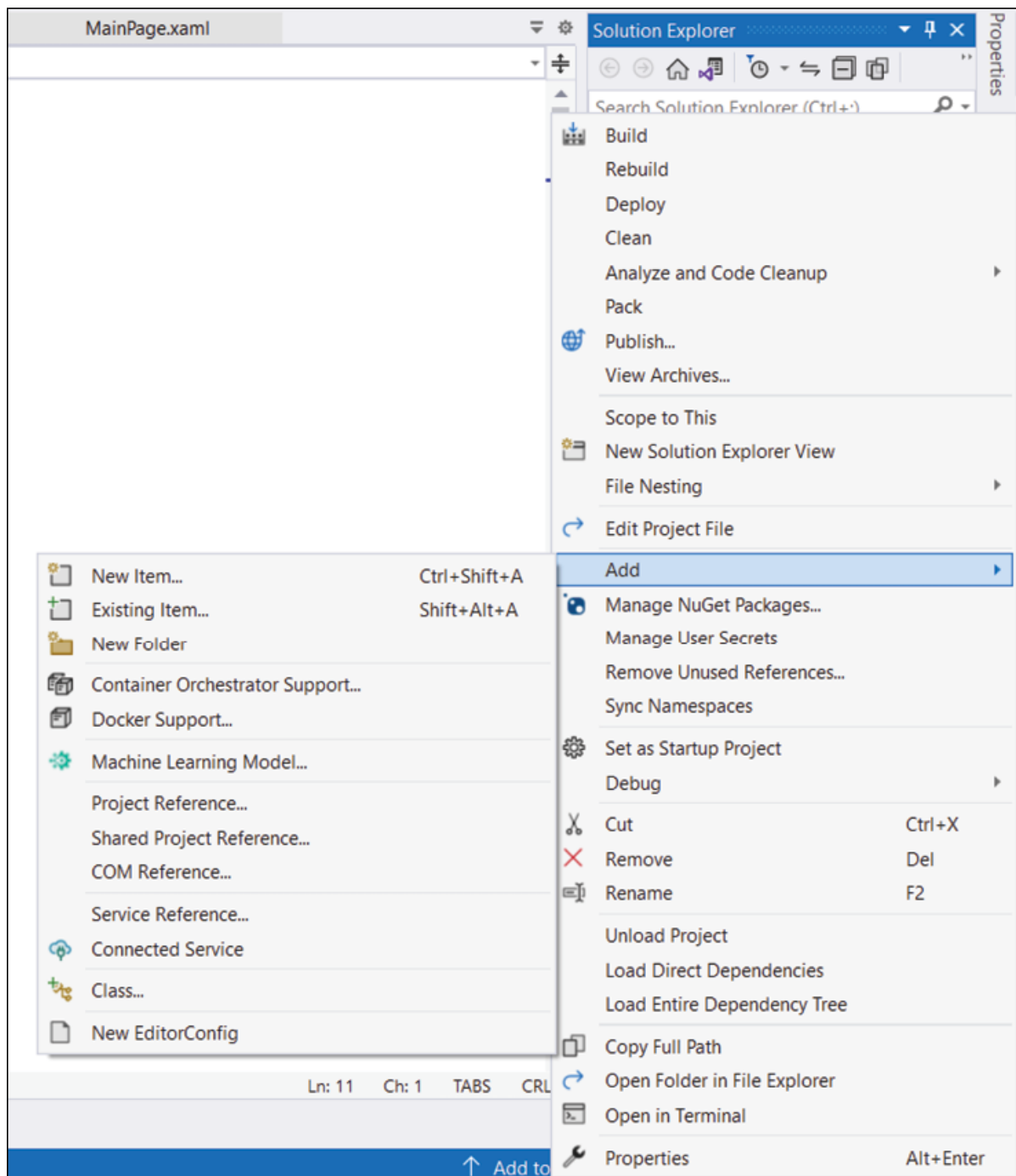


Figure 9.9: Adding a new Content Page

4. Select **.NET MAUI ContentPage (XAML)**, as shown in [Figure 9.10](#):

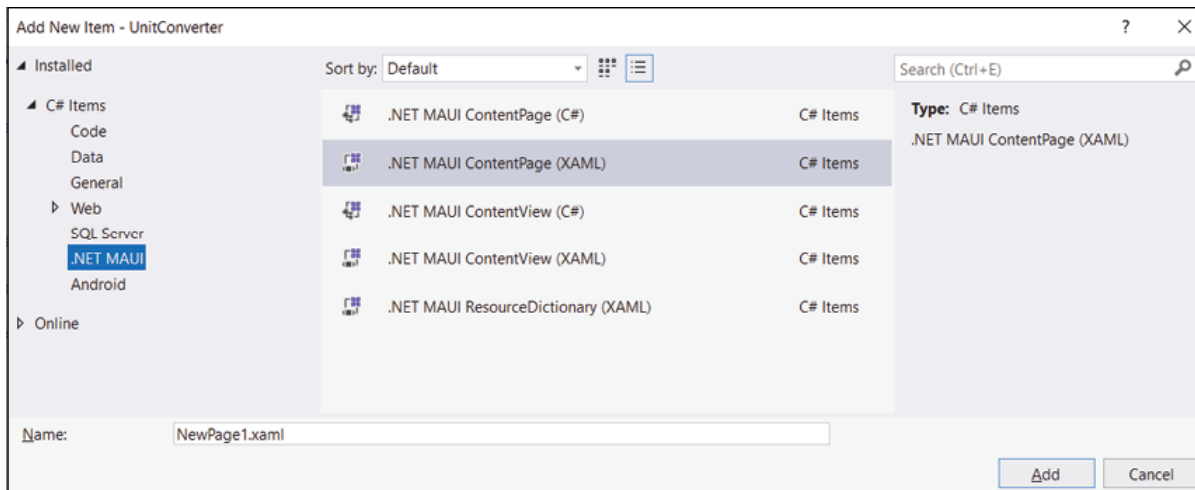


Figure 9.10: Select *.NET MAUI ContentPage (XAML)*

5. Rename the file to **LengthPage.xaml** and click **Add**. A new XAML page **LengthPage.xaml** should be added to the project and be visible in the Solution Explorer with a corresponding code-behind file named as **LengthPage.xaml.cs**.
6. To navigate from the root page (**MainPage.xaml**) to the newly created **LengthPage**, open the **MainPage.xaml.cs** file.
7. In the clicked event handler of the corresponding **Button** control, use the **Navigation.PushAsync()** method and pass an instance of the **LengthPage** as its argument. Since this is an asynchronous method, you will have to use the await keyword and Visual Studio will automatically add an async keyword to the event handler as well. Refer to the following code:

```

1.
    private async void lengthButton_Clicked(object sender, EventArgs e)
2. {
3.     await Navigation.PushAsync(new LengthPage());
4. }

```

8. Now, run the app in your emulator or device and click on the **Length** button to check if you can navigate to the content page. Observe that on the top, you will see a navigation bar with a back button. If you click the back button, it will return to the root page:

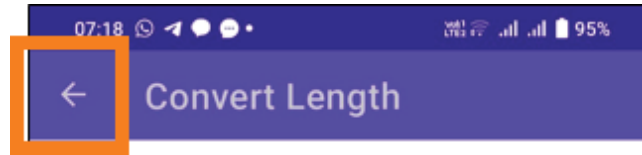


Figure 9.11: Back button to return to root page

9. Repeat *steps 3-6* to add the respective content pages for all other categories, i.e., Weight, Volume, Temperature, Area, and Time.
10. Run the app in the Emulator or device to check if the navigation system between the menu and the respective content pages is working properly.

Creating the data entry section

In the previous section, we added content pages for each type of conversion. In this section, we will create the data entry sections in the content pages, which will enable users to perform the conversion.

Let us start with the **LengthPage.xaml**. We need the user to be able to select the source unit and the target unit from the available units of measurement. For this purpose, let us add two Picker controls that can display the available units and allow the user to select the source and target units for the conversion.

1. Start by inserting a **VerticalStackLayout** in the **ContentPage** which will automatically stack its children vertically.
2. Let us add **Spacing** and **Padding** to improve the arrangement of the controls in the page. Refer to the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/d
   otnet/2021/maui"
3.             xmlns:x="http://schemas.microsoft.com
   /winfx/2009/xaml"
4.             x:Class="UnitConverter.LengthPage"
5.             Title="Convert Length">
6.     <VerticalStackLayout Spacing="20"
7.                         Padding="30,20">
8.
9.     </VerticalStackLayout>
10. </ContentPage>

```

3. Add two **Picker** controls—one to select the source unit and the other to select the target unit. We need to assign names to the **Picker** controls as **sourcePicker** and **targetPicker**, since we need to access them from the code-behind file to populate them.

4. Insert **Label** controls above each **Picker** control for describing its purpose to the user:

```
1. <Label Text="Select Source Unit" />
2. <Picker x:Name="sourcePicker" />
3. <Label Text="Select Target Unit" />
4. <Picker x:Name="targetPicker" />
```

5. Next, add an **Entry** control for the user to enter the value to be converted and an accompanying **Label** control with the instructions.

6. Since the user is expected to enter only numbers, a numeric keyboard can be shown to the user instead of the default alpha-numeric keyboard when they select the control to enter information. You can set the **Keyboard** attribute of the **Entry** to **Numeric** to show a numeric keyboard to the user. Refer to the following code:

```
1. <Label Text="Select Source Unit" />
2. <Picker x:Name="sourcePicker" />
3. <Label Text="Select Target Unit" />
4. <Picker x:Name="targetPicker" />
5. <Label Text="Enter value to convert" />
6. <Entry x:Name="sourceText"
7.     Keyboard="Numeric" />
```

7. Add a **Button** control to start the conversion. We will implement the event handler for the clicked event shortly.

8. Finally, a label control will be added to display the result of the conversion. The complete markup of the **LengthPage** is shown in the following code:

```
1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/d
   otnet/2021/maui"
3.     xmlns:x="http://schemas.microsoft.com
   /winfx/2009/xaml"
4.     x:Class="UnitConverter.LengthPage"
5.     Title="Convert Length">
```



```

6.     <VerticalStackLayout Spacing="20"
7.         Padding="30,20">
8.         <Label Text="Select Source Unit" />
9.         <Picker x:Name="sourcePicker" />
10.        <Label Text="Select Target Unit" />
11.        <Picker x:Name="targetPicker" />
12.        <Label Text="Enter value to convert" />
13.        <Entry x:Name="sourceText"
14.            Keyboard="Numeric" />
15.        <Button Text="Convert"
16.            Clicked="Button_Clicked" />
17.        <Label x:Name="resultText" />
18.    </VerticalStackLayout>
19. </ContentPage>

```

9. Remember to update the **Title** of the **ContentPage** as well. This is illustrated in the following figure:

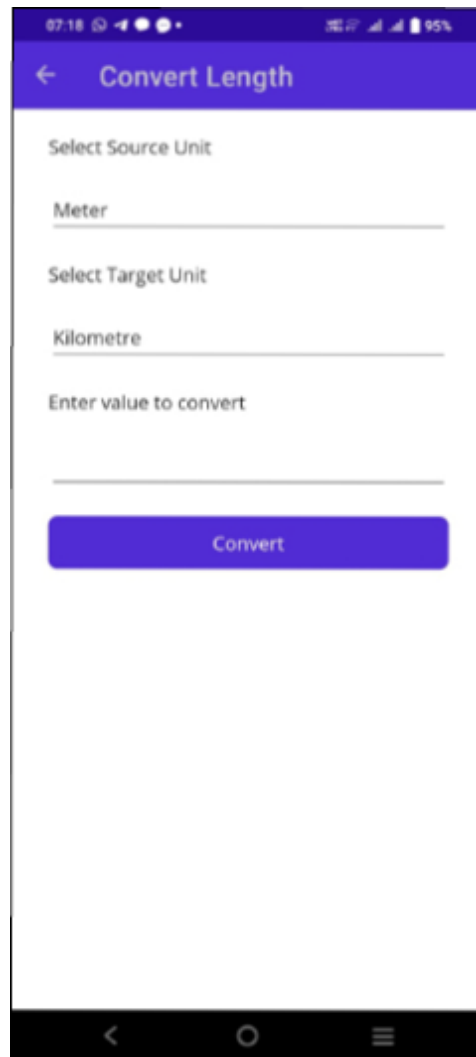


Figure 9.12: Content page for length conversion

Performing the conversions

Let us start by implementing the conversion for **Length** in the **LengthPage.xaml.cs** file:

1. We need to display the available units for conversion in the **Picker** controls for the source unit and target unit. This can be done by creating a string array that stores the units of measurement.

```
1. string[] availableUnits = { "Meter", "Kilometre",  
    "Centimetre", "Millimetre", "Mile", "Yard", "Foot", "Inch" };
```

2. Assign the array to the **ItemSource** property of the **Picker** controls.
3. For the initial state of the source unit, assign the **SelectedIndex** of the **sourcePicker** to 0 to display the first element in the array.
4. For the initial state of the target unit, we need to display the second element of the array. Use the following code to assign the **SelectedIndex** of the **targetPicker** to 1 to display the second element of the array:

```
1. namespace UnitConverter;
2.
3. public partial class LengthPage : ContentPage
4. {
5.     string[] availableUnits = { "Meter", "Kilometre",
        "Centimetre", "Millimetre", "Mile", "Yard", "Foot",
        "Inch" };
6.
7.     public LengthPage()
8.     {
9.         InitializeComponent();
10.         sourcePicker.ItemsSource = availableUnits;
11.         sourcePicker.SelectedIndex = 0;
12.         targetPicker.ItemsSource = availableUnits;
13.         targetPicker.SelectedIndex = 1;
14.     }
15.
16.     private void Button_Clicked(object sender, EventArgs e)
```

```
17.     {
```

```
18.
```

```
19.     }
```

```
20.
```

```
21. }
```

5. In the **Button_Clicked** event handler, we will perform the conversion only if two conditions are satisfied:

a. There is data entered in the **Entry** control for the source value to be converted.

b. The data entered is a valid numerical value.

c. The following code helps check if both these conditions are satisfied:

```
1. private void Button_Clicked(object sender, EventArgs e)
```

```
2. {
```

```
3.     if (!String.IsNullOrEmpty(sourceText.Text)
        && double.TryParse(sourceText.Text, out double
        sourceValue))
```

```
4. {
```

```
5. }
```

```
6. }
```

6. For the conversion, we can implement a helper method named **ConvertValue** which takes three arguments:

```
1. private double ConvertValue(double sourceValue,
    int sourceIndex, int targetIndex)
```

```
2. {
```

```
3. }
```

7. The conversion factors are mentioned in the array. Depending on the choice of the units in the two **Picker** controls, the conversion calculation is easily done using a formula. Refer to the following code:

```
1. private double ConvertValue(double sourceValue, int
   t
   sourceIndex, int targetIndex)
2. {
3.     double[] conversionFactors = { 1, 0.001, 100, 100
   0,
   0.000621371, 1.09341, 3.28084, 39.3701};
4.     return sourceValue * conversionFactors[targetIndex]
   x]
   / conversionFactors[sourceIndex];
5. }
```

8. Finally, the converted value can be displayed in the **resultLabel** using string interpolation. Refer to the following code:

```
1. if (!String.IsNullOrEmpty(sourceText.Text) &&
   double.
   TryParse(sourceText.Text, out double sourceValue))
2. {
3.     double resultValue = ConvertValue(sourceValue,
   sourcePicker.SelectedIndex, targetPicker.SelectedIndex);
4.     resultText.Text = $"Result:\n{resultValue}";
5. }
6. else
7. {
8.     resultText.Text = "";
```

9. }

9. Run the app now and check the functionality by selecting different units of measurement in the **sourcePicker** and **targetPicker** and check the converted value. This is illustrated in the following figure:

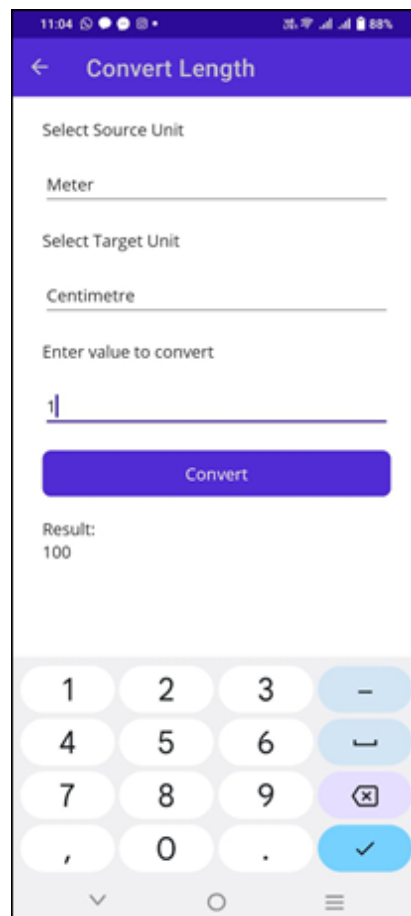


Figure 9.13: Checking the length converter

10. Repeat the above steps for other measurement categories as well, replacing the two arrays: **availableUnits** and the **conversionFactors**.
11. For weight conversion use the following code:

```
1. string[] availableUnits = { "Kilogram", "Gram", "Milligram",  
    "Tonne", "Stone", "Pound", "Ounce" };
```

2.

```
3. double[] conversionFactors = { 1, 1000, 1000000, 0
```

```
.001,  
0.157473, 2.20462, 35.274 };
```

12. For volume conversion use the following code:

```
1. string[] availableUnits = { "Litre", "Millilitre",  
    "Cubic foot", "Cubic inch", "Gallon (US)", "Gallon  
    (UK)" };
```

2.

```
3. double[] conversionFactors = { 1, 1000, 0.0353147,  
    61.0237, 0.284, 0.219};
```

13. For area conversion use the following code:

```
1. string[] availableUnits = { "Acre", "Square Meter"  
    ,  
    "Square Kilometre", "Hectare", "Square Mile", "Squ  
    are Yard",  
    "Square Foot", "Square Inch" };
```

2.

```
3. double[] conversionFactors = { 1, 4046.8564224, 0.  
    0040468564,  
    0.4046856422, 0.0015624989, 4840, 43560, 6272640 }  
    ;
```

14. For time conversion use the following code:

```
1. string[] availableUnits = { "Hour","Second", "Mill  
    isecond",  
    "Microsecond", "Minute","Day", "Week", "Month", "Y  
    ear" };
```

2.

```
3. double[] conversionFactors = { 1, 3600, 3600000, 3  
    6000000000000,  
    60, 0.0416666667, 0.005952381, 0.0013689254, 0.00
```

```
01140771 };
```

15. For temperature conversion use the following code:

```
1. string[] availableUnits = { "Celsius", "Fahrenheit", "Kelvin" };
```

16. As the conversion is non-linear, we use a formula to convert values between different units. Refer to the following code:

```
1. private double ConvertValue(double sourceValue,  
    int sourceIndex, int targetIndex)  
2. {  
3.     if (sourceIndex == targetIndex)  
4.     {  
5.         return sourceValue;  
6.     }  
7.  
8.     switch (sourceIndex)  
9.     {  
10.        case 0:  
11.            return targetIndex == 1 ? (sourceValue  
    *  
    9.0 / 5.0) + 32.0 : sourceValue + 273.15;  
12.        case 1:  
13.            return targetIndex == 0 ? (sourceValue  
    - 32.0) *  
    5.0 / 9.0 : ((sourceValue -  
    32.0) * 5.0 / 9.0) + 273.15;  
14.        case 2:  
15.            return targetIndex == 0 ? sourceValue  
    - 273.15 :
```



```

16.         default:
17.             return sourceValue;
18.     }
19. }

```

17. Run the app and check if the unit conversions between different units in each of the categories are working. It should look like the following screenshot:

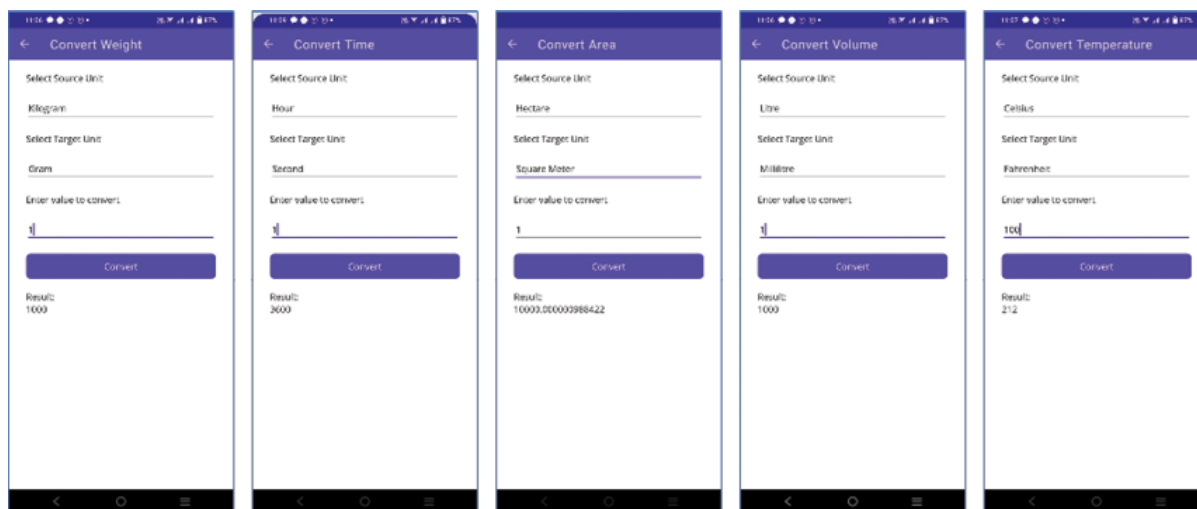


Figure 9.14: Checking unit conversions in other categories

Conclusion

In this chapter, you have developed your fourth project using .NET MAUI, and hopefully, you have been able to run it on your device as well. In case you faced any errors, check the source code provided for reference, compare it with your code and make the necessary corrections.

In the next chapter, we will develop another app with advanced capabilities including accessing a REST API.

Points to remember

Here are a few key takeaways from this chapter:

- Ensure that you choose the right .NET Framework version while creating the project.
- This app demonstrated how you can implement a multi-page app with a navigation system in .NET MAUI.
- The root page must be set in the App class.
- To navigate from the root page to the content pages, use the `PushAsync()` method in the Navigation class. Remember to use the `await` keyword and mark the wrapper function as an async method as well.
- The focus of this chapter was on .NET MAUI features and not on design patterns. For real-world implementation, you can explore how to use the MVVM pattern, which provides several advantages, including separation of concerns, testability, and easy maintainability. It is beyond the scope of this book.
- If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Project-5: Weather App

Introduction

Despite all the advancements in science and technology, there are some things in nature that are beyond our control. Weather is one such aspect that affects our lives in many ways. It decides what we do, wear, carry, and how we travel. A weather app is, therefore, a trusted companion in helping us navigate different weather conditions and ensure that our lives are not affected. Besides knowing about the weather and the forecast, the weather app provides an excellent learning opportunity, as it involves some of the advanced skills interfacing with REST API, parsing data, and presenting it in a user-friendly format.

In this chapter, we will develop a simple weather app that will allow the user to enter the name of a place and find out the weather, particularly the current temperature and type of weather, and know the weather forecast for the next seven days, including max and min. temperature. In the process, we will learn how to work with REST API and parse the data. We will also learn how to use the **Model-View-ViewModel (MVVM)** architectural pattern in application development.

Structure

In this chapter, we will discuss the following topics:

- Introduction to MVVM architecture
- Creating the project structure and the UI

- Using the weather API
- Calling the weather API from code
- Creating custom type converters
- Updating the UI with weather data
- Displaying seven-day weather forecast

Objectives

After reading this chapter, you will be able to:

- Develop an application using MVVM architecture.
- You will also be able to work with REST API and calling the API from code.
- Additionally, you will learn about Parsing JSON data obtained from REST API, and advanced Data Binding skills for binding the UI to data.
- By the end of this chapter, you will learn how to implement different types of type converters and work with CollectionView and ObservableCollection.

Introduction to MVVM architecture

In this project, we will implement what is known as the MVVM architecture. It is a very popular architectural pattern used in .NET MAUI projects. It makes your code more manageable and maintainable.

In the MVVM architectural pattern, the application code is divided into three components:

Model, View and ViewModel.

- **Model:** This is where you define the structure of the data in the form of classes and properties.
- **View:** This is what the user sees and interacts with. It consists of the UI controls like the Buttons, Labels, Textboxes, Images, etc.
- **ViewModel:** This is the bridge between the Model and the View. It takes data based on the model and prepares it in a way that the View can understand it. It also handles user interactions and sends that information back to the Model.

The main advantages of MVVM architecture includes the following:

- **Separation of concerns:** As the application code is separated into the three components, i.e., model, ViewModel and view, it is easier to manage and update.
- **Testability:** MVVM makes it easier to test your app's functionality because you can test the ViewModel independently from the user interface.
- **Maintainability:** As your app evolves, MVVM makes it easier to add new features or make changes without breaking the existing code.

Creating the project structure and the UI

Follow these steps to create a new app for the weather app:

1. Start Visual Studio.
2. In the dialog that appears, select the option, **Create a new project**, as shown in *Figure 10.1*:

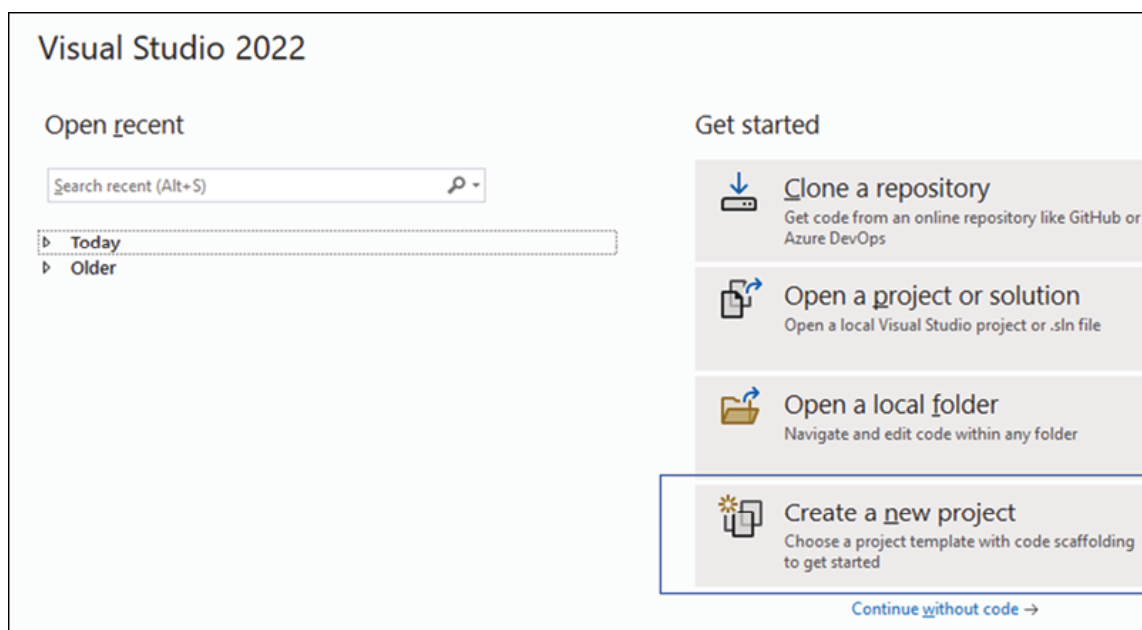


Figure 10.1: Creating a new project in Visual Studio

3. Select the programming language as C# and MAUI project to view the MAUI project templates, as shown in *Figure 10.2*:

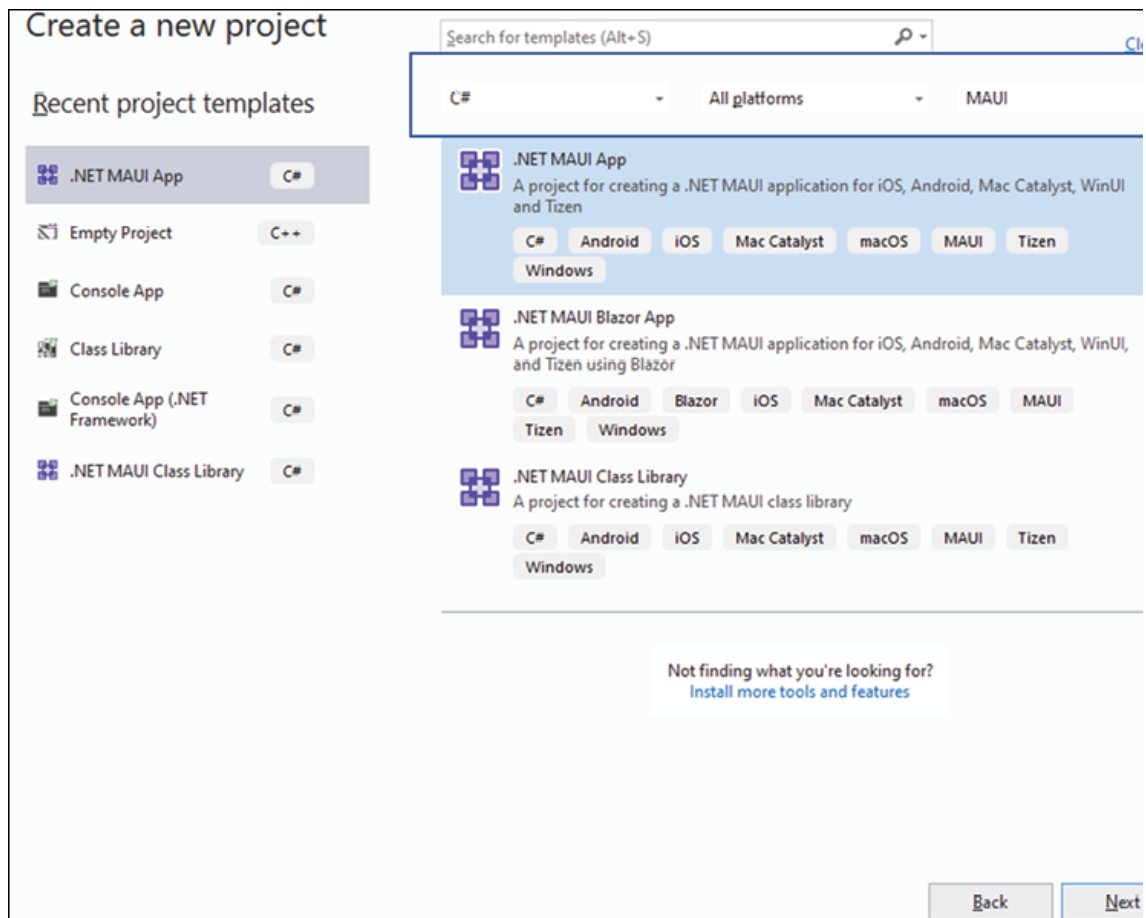


Figure 10.2: Selecting .NET MAUI project template

4. Select **.NET MAUI App** and click on **Next**.
5. Enter the project name as **WeatherApp**. Select the preferred destination folder or leave it with the default setting, as shown in [Figure 10.3](#):

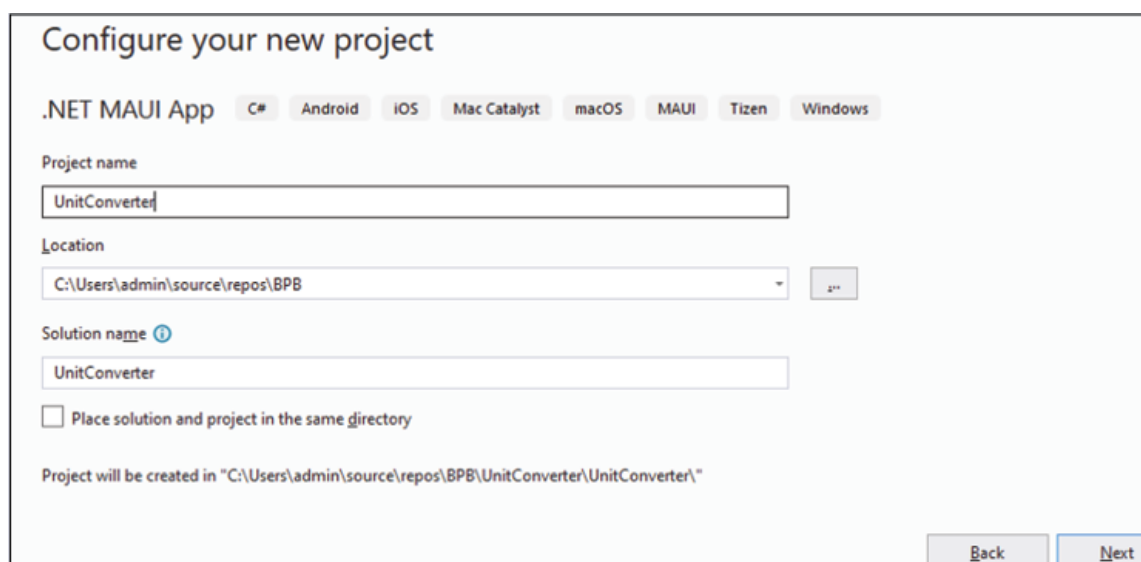


Figure 10.3: Configure your project

6. You can choose the latest Framework version which is displayed under **Additional Information** and click on **Create**, as shown in [Figure 10.4](#):

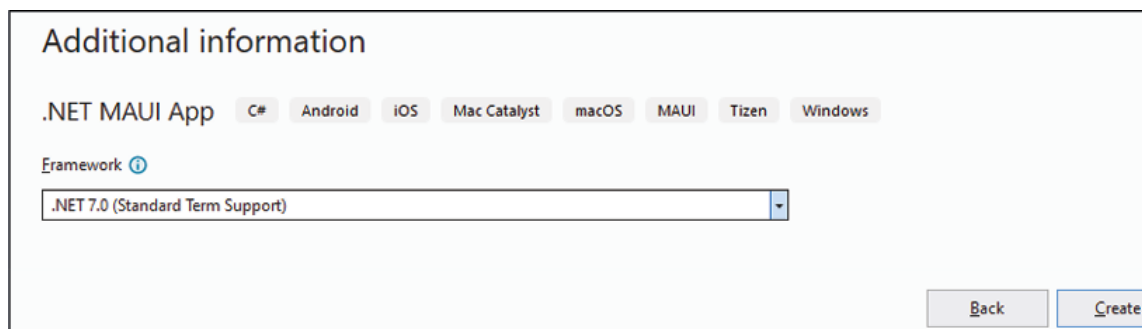


Figure 10.4: Selecting the framework version

7. The MAUI project is created and within a few minutes, you should be able to see the project in the **Solution Explorer** as shown, in [Figure 10.5](#):

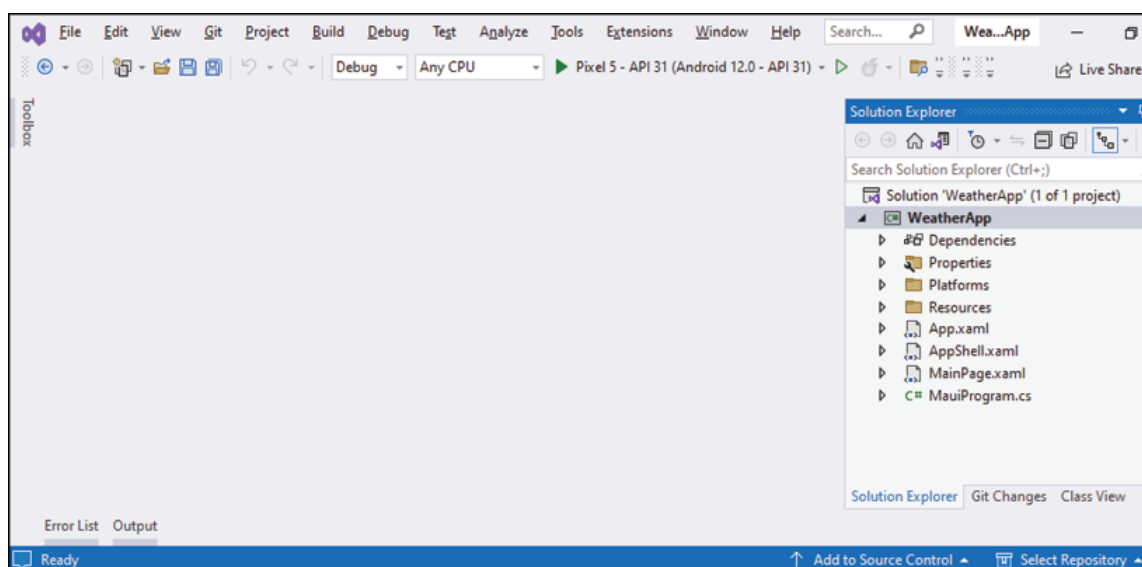


Figure 10.5: Default solution in Visual Studio

8. Once the project is created, got to **Solution Explorer** and right click on the project and add a new folder named **MVVM**.
9. Within the MVVM folder, create sub-folders for the Models, Views and ViewModels.
10. In the **Models** folder, create an empty class called **WeatherData.cs**. We will edit this file later.
11. In the **ViewModels** folder, create a class called **WeatherViewModel.cs**.

12. In the **Views** folder, add a XAML file and name it as **WeatherView.xaml**. This is illustrated in the following figure:

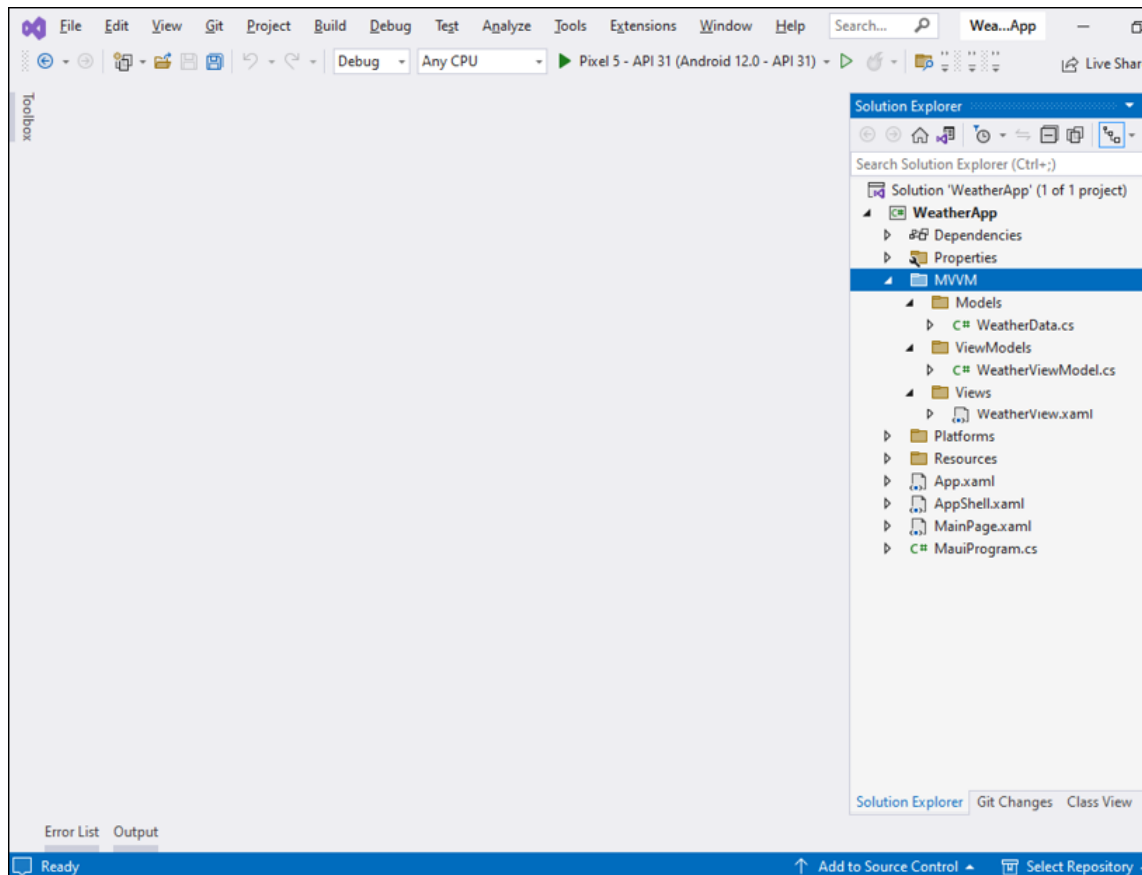


Figure 10.6: MVVM folder structure

13. Create a new instance of **WeatherViewModel** and assign the **BindingContext** to this instance:

```
1. public partial class WeatherView : ContentPage
2. {
3.     public WeatherView()
4.     {
5.         InitializeComponent();
6.         BindingContext = new WeatherViewModel();
7.     }
8. }
```


Designing the user interface

To design the UI, follow these steps:

1. We need to change the startup page that will be displayed when the app is launched. In **app.xaml.cs**, make the following changes. Refer to the following code:

```
1. public partial class App : Application
2. {
3.     public App()
4.     {
5.         InitializeComponent();
6.         MainPage = new WeatherView();
7.     }
8. }
```

2. In **WeatherView.xaml**, add a **ScrollView** as a child of the **ContentPage**.
3. Inside the **ScrollView**, add a **VerticalStackLayout** which will help organize the child controls in a one-dimensional vertical stack.
4. For better alignment, let us add a spacing of 20 between each child control, and a padding of 10 on the right and left side between the contents and its edge. Refer to the following code:

```
1. <VerticalStackLayout Spacing="20"
2.                     Padding="10,0"
3.                     VerticalOptions="StartAndExpand"
4. >
```

5. In our app, we need to provide an option to search for the weather for any city. We can use a **SearchBar** for this purpose. The **SearchBar** also provides a **Placeholder** where we can provide the help text for the user. Refer to the following code:

```
1. <SearchBar x:Name="searchBar"
```

2. `Placeholder="Search" />`
6. Let us add the UI controls that will be used to display the weather data. Add Label controls to display the date, place name, weather type, and the current temperature. Also, add an Image control to display the weather status visually. Refer to the following code:
 1. `<Label Text="Oct 30, 2023"`
 2. `FontAttributes="Bold"`
 3. `FontSize="Medium"`
 4. `HorizontalOptions="Center" />`
 5. `<Label Text="Bangalore"`
 6. `HorizontalOptions="Center" />`
 7. `<Image Source="test.png"`
 8. `HeightRequest="200"`
 9. `HorizontalOptions="Center" />`
 10. `<Label Text="20 °C"`
 11. `HorizontalOptions="Center" />`
 12. `<Label Text="Clear Sky"`
 13. `HorizontalOptions="Center" />`
7. Save any random image file and with the file extension **.png** in the Resources/Images folder and rename it to **test.png**. This will serve as a placeholder for now.
8. Run the app in your emulator/device to test if the screen rendered is ok and make any adjustments to the XAML markup, if required.

Updating the UI with date

To update the UI with date, follow these steps:

1. To display the date, we can bind the label to a **DateTime** property defined in the **ViewModel**, which will display the current date. Refer to the following code:

```

1. public class WeatherViewModel
2. {
3.     public DateTime Date { get; set; } = DateTime
       .Now;
4. }

```

- Now, in the **WeatherView.xaml**, you can bind the label's **Text** property to this value using the databinding expression. Use the appropriate formatting specifier to display the date in your preferred format. With the following markup, the date will be formatted as month day, year (e.g, September 09, 2024). Refer to the following code:

```

1. <Label Text="
   {Binding Date, StringFormat='{0:MMMM dd, yyyy}}'"
   />

```

Updating the UI with place name

To display the place name, we can use the text entered by the user in the **SearchBar**. Follow these steps to do so:

- When the user submits the place name in the search bar, we must invoke a function in the code-behind file. To implement this, we can use the **SearchCommand** property and bind it to an **ICommand** defined in the **ViewModel**. Refer to the following code:

```

1. <SearchBar x:Name="searchBar"
2.           Placeholder="Search"
3.           SearchCommand="{Binding SearchCommand}"
4.           SearchCommandParameter="{Binding Source=
       {x:Reference searchBar}, Path=Text}"
5.           VerticalOptions="Center" />

```

- The string entered by the user can be passed to **ICommand** using the **SearchCommandParameter** property. Refer to the following code:

```

1. public ICommand SearchCommand =>
   new Command(async (searchText) =>

```

```

2. {
3.     // TO DO
4. });

```

3. For displaying the name of the location, let us create a property in the ViewModel to represent the place. Refer to the following code:

```

1. public string PlaceName { get; set; }

```

4. The **PlaceName** can be populated using the value entered by the user in the **SearchBar**. Refer to the following code:

```

1. public ICommand SearchCommand =>
2.     new Command(async (searchText) =>
3.     {
4.         PlaceName = searchText.ToString();
5.
6.         // .. TO DO
7.
8.     });

```

Using the weather API

To get the weather data, we can use the free open-source weather API. Click here to access the API docs: <https://open-meteo.com/en/docs>

Uncheck the hourly weather variables if they are selected. Select only the following:

Current weather:

- Temperature (2 m)
- Weathercode

Daily weather variables:

- Weathercode
- Maximum temperature (2 m)

- Minimum temperature (2 m)

By default, the API provides forecasts for 7 days and the temperature unit is Celsius, as shown in the following figure:

The screenshot shows the Open-Meteo website's configuration interface. The browser address bar displays a URL with various parameters. The page title is '15-Minutely Weather Variables'. On the left, a sidebar menu lists categories like 'Weather Forecast', 'Historical Weather', and 'Ensemble Models'. The main content area has four sections: 'Daily Weather Variables' with a list of checkboxes for temperature, precipitation, and wind; 'Current Weather' with similar checkboxes; 'Settings' with four dropdown menus for units and time format; and a 'Note' at the bottom stating that current conditions are based on 15-minute model data.

Figure 10.7: Weather API settings

For daily variables, a time zone needs to be set as per your desired location.

Under **Preview and API URL**, a unique API URL is created based on your settings, as shown in the following figure:

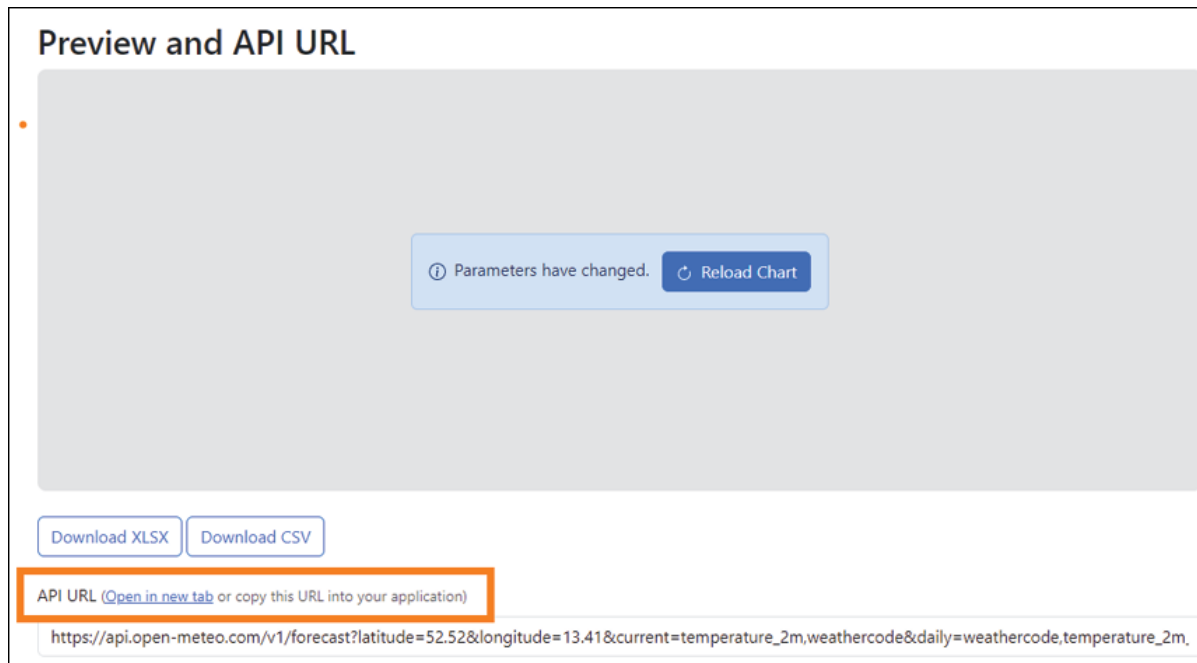


Figure 10.8: Weather API preview

Creating the model

To populate the weather data in the UI, we need to create a model class with properties for each parameter provided by the weather API. Fortunately, Visual Studio provides an easy way to do this.

1. Open the API URL in a new tab to see the JSON data returned by the API. This is illustrated in the following figure:

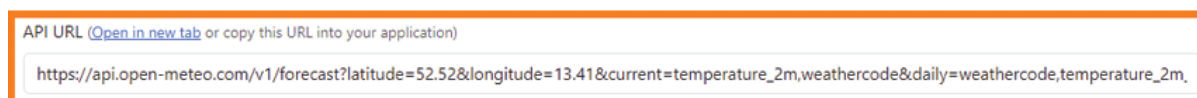


Figure 10.9: Weather API URL

2. Copy the JSON data to your clipboard:

```
{"latitude":52.52,"longitude":13.419998,"generationtime_ms":0.07998943328857422,"utc_offset_seconds":25200,"time zone":"Asia/Bangkok","timezone_abbreviation":"+07","elevation":38.0,"current_units":{"time":"iso8601","interval":"seconds","temperature_2m":"°C","weathercode":"wmo code"},"current":{"time":"2023-10-30T07:30","interval":900,"temperature_2m":11.8,"weathercode":3},"daily_units":{"time":"iso8601","weathercode":"wmo code","temperature_2m_max":"°C","temperature_2m_min":"°C"},"daily":{"time":["2023-10-30",
```

```
"2023-10-31", "2023-11-01", "2023-11-02", "2023-11-03", "2023-11-04",
"2023-11-05"], "weathercode": [80, 80, 61, 3, 61, 61, 61],
"temperature_2m_max":
[14.1, 12.8, 13.7, 14.2, 12.1, 10.9, 12.7],
"temperature_2m_min": [10.3, 10.8, 7.8, 8.1, 9.2, 7.1, 9.8]]}}
```

3. Open the **WeatherData.cs** file that you created earlier in the **Models** folder.
4. Click on **Edit | Paste Special** and select **Paste JSON as Classes** as shown in the following figure:

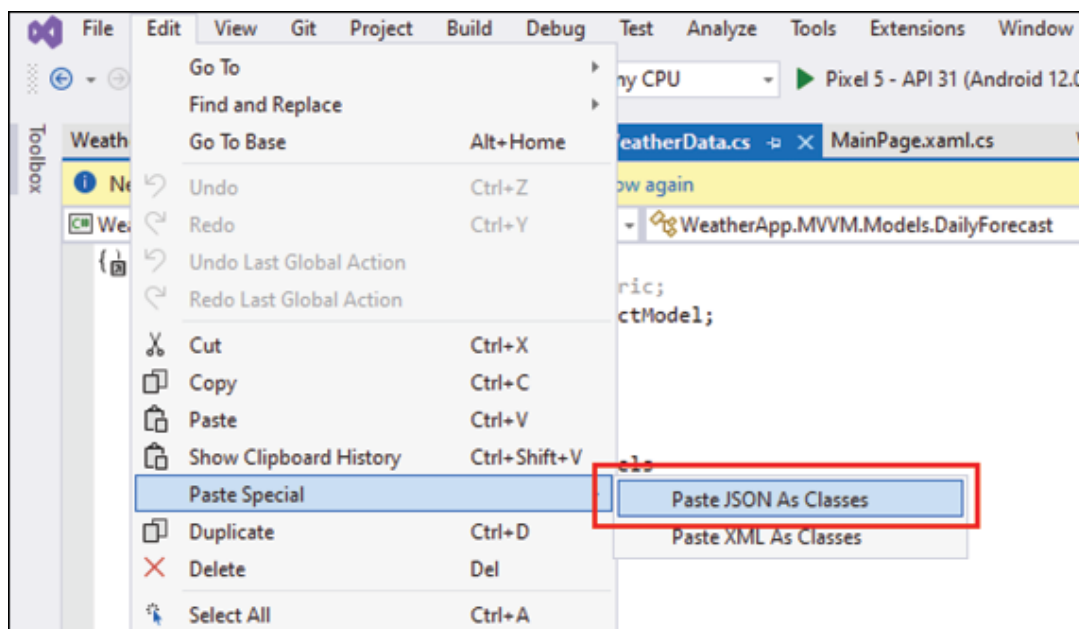


Figure 10.10: Paste as JSON Classes in Visual Studio

5. Visual Studio will automatically create classes and properties based on the JSON data that you copied. Rename the **Rootobject** class to **WeatherData** as shown in the following code:

```
1. namespace WeatherApp.MVVM.Models
2. {
3.
4.     public class WeatherData
5.     {
6.         public float latitude { get; set; }
7.         public float longitude { get; set; }
8.         public float generationtime_ms { get; set; }
9.         public int utc_offset_seconds { get; set; }
10.        public string timezone { get; set; }
```

```

11.         public string timezone_abbreviation { get; se
t; }
12.         public float elevation { get; set; }
13.         public Current_Units current_units { get; set
; }
14.         public Current current { get; set; }
15.         public Daily_Units daily_units { get; set; }
16.         public Daily daily { get; set; }
17.     }
18.
19.     public class Current_Units
20.     {
21.         public string time { get; set; }
22.         public string interval { get; set; }
23.         public string temperature_2m { get; set; }
24.         public string weathercode { get; set; }
25.     }
26.
27.     public class Current
28.     {
29.         public string time { get; set; }
30.         public int interval { get; set; }
31.         public float temperature_2m { get; set; }
32.         public int weathercode { get; set; }
33.     }
34.
35.     public class Daily_Units
36.     {
37.         public string time { get; set; }
38.         public string weathercode { get; set; }
39.         public string temperature_2m_max { get; set;
}
40.         public string temperature_2m_min { get; set;
}
41.     }
42.
43.     public class Daily
44.     {

```



```

45.         public string[] time { get; set; }
46.         public int[] weathercode { get; set; }
47.         public float[] temperature_2m_max { get; set;
    }
48.         public float[] temperature_2m_min { get; set;
    }
49.     }
50.
51. }

```

Calling the weather API from code

Let us examine the weather API URL that we created in the previous step:

[https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41¤t=temperature_2m,weathercode&daily=weathercode,temperature_2m_max,temperature_2m_min&time zone=auto](https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41¤t=temperature_2m,weathercode&daily=weathercode,temperature_2m_max,temperature_2m_min&time_zone=auto)

Observe that some random values of latitude and longitude were selected by default. We have to replace these with the actual latitude and longitude values of the place that the user submits in the **SearchBar**.

To get the latitude and longitude values of a place, we can use the following code:

```

1. private async Task<Location> GetCoordinatesAsync(string
    address)
2. {
3.     IEnumerable<Location> locations = await Geocoding.De
    fault.GetLocationsAsync(address);
4.     Location location = locations?.FirstOrDefault();
5.     if (location != null)
6.         Console.WriteLine($"Latitude: {location.Latitude
    }, Longitude: {location.Longitude}, Altitude: {location.
    Altitude}");
7.     return location;
8. }

```

We can call the above function from the **SearchCommand** that we partially implemented earlier. Refer to the following code:

```

1. public ICommand SearchCommand =>
2.     new Command(async (searchText) =>

```

```

3.     {
4.         PlaceName = searchText.ToString();
5.         var location = await GetCoordinatesAsync(search
Text.ToString());
6.         // .. TO DO
7.     });

```

The **Location** class is defined as a part of the **Microsoft.Maui.Devices.Sensors** namespace and has properties to represent the Latitude and Longitude information. You can extract these values from the location object and provide it in the weather API to get the weather data for that location.

https://api.open-meteo.com/v1/forecast?latitude={location.Latitude}&longitude={location.Longitude}¤t=temperature_2m,weathercode&daily=weathercode,temperature_2m_max,temperature_2m_min&timezone=auto

To call the weather API from code, we can use the **HttpClient** class provided in the .NET Framework. Refer to the following code:

```

1. private async Task GetWeather(Location location)
2. {
3.     var url = $"https://api.open-meteo.com/v1/forecast?
latitude={location.Latitude}&longitude=
{location.Longitude}&current=temperature_2m,weathercode&
daily=weathercode,
temperature_2m_max,temperature_2m_min&timezone=auto";
4.
5.     var response = await client.GetAsync(url);
6.
7.     if (response.IsSuccessStatusCode)
8.     {
9.         using (var responseStream = await response.Conte
nt.ReadAsStreamAsync())
10.        {
11.            var data = await JsonSerializer.DeserializeA
sync<WeatherData>(responseStream);
12.            Weather = data;
13.        }
14.    }
15. }

```

Notice that we have declared a property in the ViewModel to store the **Weather** data:

```
1. public WeatherData Weather { get; set; }
```

We can call the above function from the **SearchCommand** that we partially implemented earlier:

```
1. public ICommand SearchCommand =>
2.     new Command(async (searchText) =>
3.     {
4.         PlaceName = searchText.ToString();
5.         var location = await GetCoordinatesAsync(searchText.ToString());
6.         await GetWeather(location);
7.     });
```

Creating custom type converters

For the current weather, we get the temperature value in Celsius which can be used directly. However, the weather description is returned as a **WMO Weather interpretation code (WW)**.

The following table shows various WWs:

Code	Description
0	Clear sky
1, 2, 3	Mainly clear, partly cloudy, and overcast
45, 48	Fog and depositing rime fog
51, 53, 55	Drizzle: Light, moderate, and dense intensity
56, 57	Freezing drizzle: Light and dense intensity
61, 63, 65	Rain: Slight, moderate and heavy intensity
66, 67	Freezing rain: Light and heavy intensity
71, 73, 75	Snow fall: Slight, moderate, and heavy intensity

77	Snow grains
80, 81, 82	Rain showers: Slight, moderate, and violent
85, 86	Snow showers slight and heavy
95 *	Thunderstorm: Slight or moderate
96, 99 *	Thunderstorm with slight and heavy hail

Table 10.1 : WMO weather interpretation codes

To make our app user-friendly, we can display the weather description in text and represent it with an appropriate image in the UI. For this, we have to implement two converters:

- **WeatherCodeToStringConverter**: To convert the weather code to text
- **WeatherCodeToImageConverter**: To convert the weather code to image

To create converters, follow these steps:

1. Create a new folder and name it as **Converters**.
2. Add a class named **WeatherCodeToStringConverter** and implement the **IValueConverter** interface. The **IValueConverter** interface implements two methods **Convert** and **ConvertBack**. We only need the **Convert** method which can be implemented as shown in the following code, to return the corresponding description against each **Weathercode**:

```

1. public class WeatherCodeToStringConverter : IValueConverter
2. {
3.     public object Convert(object value, Type targetType,
4.         object parameter, CultureInfo culture)
5.     {
6.         var code = (int)value;
7.         switch (code)
8.         {
9.             case 0: return "Clear sky";

```

```

10.         case 1: return "Mainly clear";
11.         case 2: return "Partly cloudy";
12.         case 3: return "Overcast";
13.         case 45: return "Fog";
14.         case 48: return "Depositing rime fog";
15.         case 51: return "Drizzle: Light intensity
";
16.         case 53: return "Drizzle: Moderate intens
ity";
17.         case 55: return "Drizzle: Dense intensity
";
18.         case 56: return "Freezing Drizzle: Light
intensity";
19.         case 57: return "Freezing Drizzle: Dense
intensity";
20.         case 61: return "Rain: Slight intensity";
21.         case 63: return "Rain: Moderate intensity
";
22.         case 65: return "Rain: Heavy intensity";
23.         case 66: return "Freezing Rain: Light int
ensity";
24.         case 67: return "Freezing Rain: Heavy int
ensity";
25.         case 71: return "Snow fall: Slight intens
ity";
26.         case 73: return "Snow fall: Moderate inte
nsity";
27.         case 75: return "Snow fall: Heavy intensi
ty";
28.         case 77: return "Snow grains";
29.         case 80: return "Rain showers: Slight";
30.         case 81: return "Rain showers: Moderate";
31.         case 82: return "Rain showers: Violent";
32.         case 85: return "Snow showers: Slight";
33.         case 86: return "Snow showers: Heavy";
34.         case 95: return "Thunderstorm: Slight or
moderate";
35.         case 96:

```

```

36.         case 99: return "Thunderstorm with slight
           and heavy hail";
37.         default: return "Not defined";
38.     }
39. }
40. }

```

3. In order to represent the weather visually, we can use icons to represent the weather status. There are several websites which provide free icons that can be used. We will use the icons from www.iconfinder.com. Download them and add them to the Resources/Images folder as shown in the following figure:

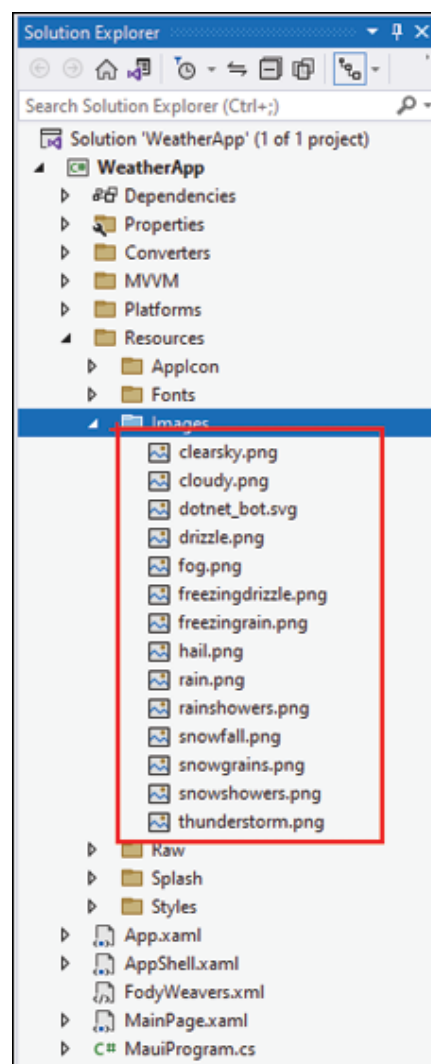


Figure 10.11: Weather icons in the images folder

You can also download icons from other websites or create your own, provided they are in formats compatible with XAML.

For your reference, the thumbnail images of the icons that we added are displayed in the following figure:

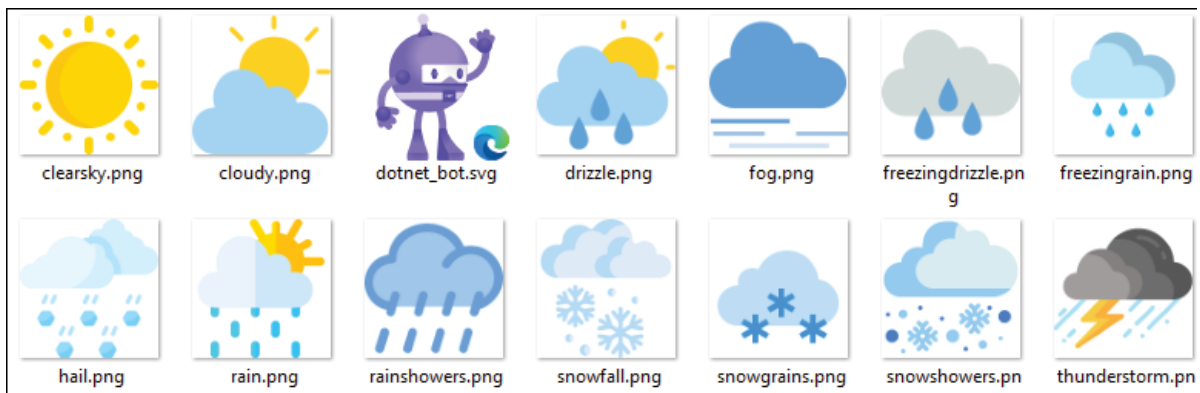


Figure 10.12: Weather icons—Thumbnail view

4. Add another class and name it as **WeatherCodeToImageConverter**.
5. Implement the **IValueConverter** interface and the **Convert** method as shown in the following code:

```

1. public class WeatherCodeToImageConverter : IValueConverter
2. {
3.     public object Convert(object value, Type targetType,
4.         object parameter, CultureInfo culture)
5.     {
6.         var code = (int)value;
7.         switch (code)
8.         {
9.             case 0: return "clearsky.png";
10.            case 1: return "cloudy.png";
11.            case 2: return "cloudy.png";
12.            case 3: return "cloudy.png";
13.            case 45: return "fog.png";
14.            case 48: return "fog.png";
15.            case 51: return "drizzle.png";
16.            case 53: return "drizzle.png";
17.            case 55: return "drizzle.png";
18.            case 56: return "freezingdrizzle.png";
19.            case 57: return "freezingdrizzle.png";

```

```

20.         case 61: return "rain.png";
21.         case 63: return "rain.png";
22.         case 65: return "rain.png";
23.         case 66: return "freezingrain.png";
24.         case 67: return "freezingrain.png";
25.         case 71: return "snowfall.png";
26.         case 73: return "snowfall.png";
27.         case 75: return "snowfall.png";
28.         case 77: return "snowgrains.png";
29.         case 80: return "rainshowers.png";
30.         case 81: return "rainshowers.png";
31.         case 82: return "rainshowers.png";
32.         case 85: return "snowshowers.png";
33.         case 86: return "snowshowers.png";
34.         case 95: return "thunderstorm.png";
35.         case 96: return "hail.png";
36.         default: return "";
37.     }
38. }
39. }

```

Updating the UI with weather data

To update the weather data, follow these steps:

1. In **WeatherView.xaml**, instantiate the converters using the following code:

```

1. <?xml version="1.0" encoding="utf-8" ?>
2.
   <ContentPage xmlns="http://schemas.microsoft.com/dotn
   et/2021/maui"
3.
   xmlns:x="http://schemas.microsoft.com/wi
   nfx/2009/xaml"
4.
   x:Class="WeatherApp.MVVM.Views.WeatherVi
   ew"
5.
   xmlns:converter="clr-
   namespace:WeatherApp.Converters"

```



```

6.         Title="WeatherView">
7.     <ContentPage.Resources>
8.
9.         <converter:WeatherCodeToStringConverter x:Key=
            =
            "WeatherCodeToStringConverter" />
10.        <converter:WeatherCodeToImageConverter x:Key=
            "WeatherCodeToImageConverter" />
10.    </ContentPage.Resources>

```

2. To display the weather code as an image, use the following markup:

```

1. <Image Source="
    {Binding Weather.current.weathercode,
    Converter=
    {StaticResource WeatherCodeToImageConverter}}"
2.     HeightRequest="200"
3.     HorizontalOptions="Center" />

```

3. To display the weather code description, use the following markup:

```

1. <Label Text="{Binding Weather.current.weathercode,
    Converter=
    {StaticResource WeatherCodeToStringConverter}}"
2.     HorizontalOptions="Center" />

```

4. Displaying the temperature is easy. You just have to bind the **Label** control to the **Weather.current.temperature_2m** property. Refer to the following code:

```

1. <Label Text="
    {Binding Weather.current.temperature_2m, StringFormat
    ='{0} °C'}"
2.     HorizontalOptions="Center" />

```

5. Run the application and enter a place name (e.g., Bangalore) and check if the weather data is displayed.
6. The **ViewModel** class must implement **INotifyPropertyChanged** interface to ensure that the UI updates and reflects the weather data if the user submits successive changes to the place name. Instead of implementing the **INotifyPropertyChanged** interface in the traditional way, we can make use of the **PropertyChanged.Fody** package which helps generate the boilerplate code and inject the necessary event and event-invokers into the class at run time.

7. Further, click on **Tools | NuGet package manager | Manage NuGet Packages for Solution | Select Browse** and search for **Fody** as shown in the following figure:

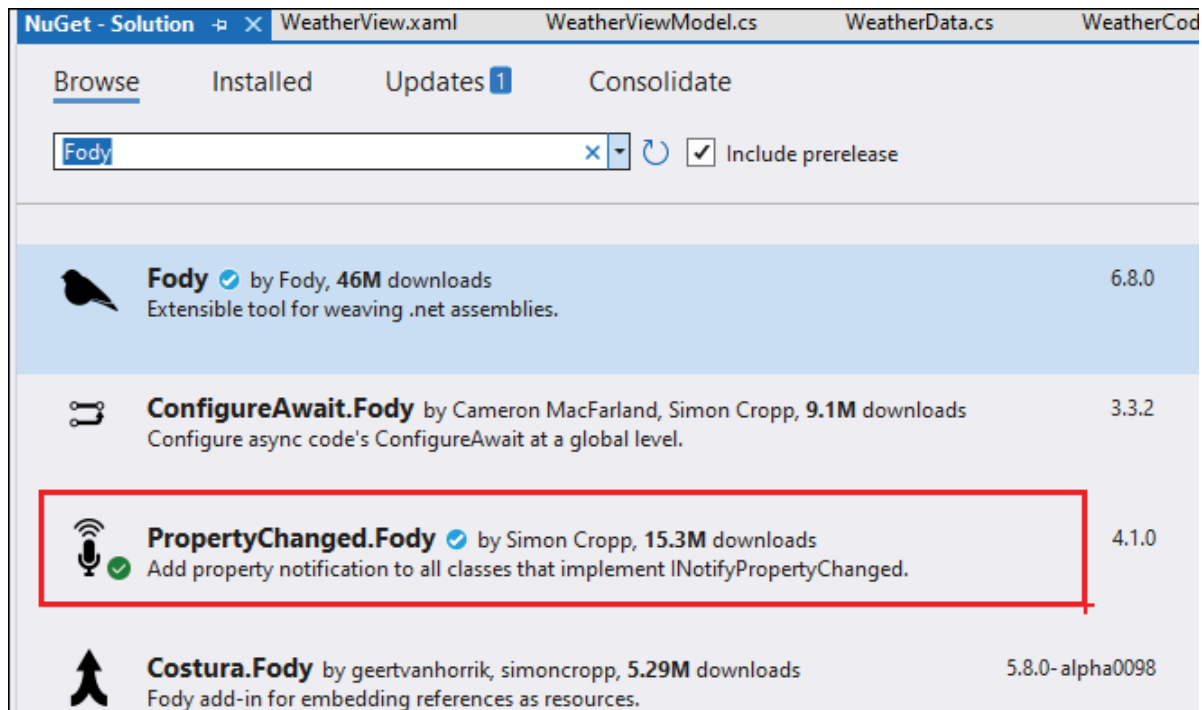


Figure 10.13: Installing PropertyChanged.Fody Package

8. Select **PropertyChanged.Fody** and install it.
9. Add an attribute [**AddINotifyPropertyChangedInterface**] to the **ViewModel** class:
1. [**AddINotifyPropertyChangedInterface**]
 2. **public class** WeatherViewModel
10. Run the app now and check if successive searches with different place names are reflecting changes in the UI.

Displaying seven-day weather forecast

The weather API also provides the weather forecast for seven days, by default. To retrieve and store this data, we can create a new class in our model that stores the weather data for each day. To do this, follow these steps:

1. Add the following class to the **WeatherData.cs** file:

```
1. public class DailyForecast
2. {
```

```

3.     public string time { get; set; }
4.     public int weathercode { get; set; }
5.     public float temperature_2m_max { get; set; }
6.     public float temperature_2m_min { get; set; }
7. }

```

2. Add a property of type **ObservableCollection<DailyForecast>** in the **WeatherData** class which will be used to store the weather forecast data retrieved from the weather API. Refer to the following code:

```

1. public ObservableCollection<DailyForecast> dailyForecasts { get; set; } = new ObservableCollection<DailyForecast>();

```

3. In the **ViewModel**, modify the **GetWeather()** method that we implemented earlier to retrieve the weather forecast data and store it in the **ObservableCollection** that we created. Refer to the following code:

```

1. private async Task GetWeather(Location location)
2. {
3.     var url = $"https://api.open-meteo.com/v1/forecast?latitude={location.Latitude}&longitude={location.Longitude}&current=temperature_2m,weathercode&daily=weathercode,temperature_2m_max,temperature_2m_min&timezone=auto";
4.
5.     var response = await client.GetAsync(url);
6.
7.     if (response.IsSuccessStatusCode)
8.     {
9.         using (var responseStream = await response.Content.ReadAsStreamAsync())

```

```

10.         {
11.             var data = await JsonSerializer.DeserializeAsync
<WeatherData>(responseStream);
12.             Weather = data;
13.
14.             Debug.WriteLine($"*****{Weather.
current.temperature_2m}
{Weather.current.weathercode}");
15.
16.             for (int i = 0; i < Weather.daily.time.Length; i++)
17.             {
18.                 var dailyForecast = new DailyForecast
{
19.                     time = Weather.daily.time[i],
20.                     temperature_2m_max = Weather.daily.
temperature_2m_max[i],
21.                     temperature_2m_min = Weather.daily.
temperature_2m_min[i],
22.                     weathercode = Weather.daily.weathercode[i]
23.                 };
24.
25.                 Weather.dailyForecasts.Add(dailyForecast);
26.             }
27.

```

28. }

29. }

30. }

Updating the weather forecast

To display the Weather forecast in the UI, we can use a **CollectionView** and bind its **ItemsSource** property to the **ObservableCollection** that stores the weather forecast data.

The **CollectionView** needs to be told how to display the data it has, using its **ItemTemplate** property. The **DataTemplate** is used to define how each item will be represented. We will define a **Grid** with three rows and two columns to display the data for each day. The first column will be used to display the icon corresponding to the Weathercode spanning all the three rows. In the second column, the weather data will be displayed in three rows. Refer to the following code:

```
1. <CollectionView x:Name="weatherforecast"
2.                ItemsSource="
   {Binding Weather.dailyForecasts}">
3.     <CollectionView.ItemTemplate>
4.         <DataTemplate>
5.             <Grid BackgroundColor="Beige"
6.                 Margin="10"
7.                 HeightRequest="100">
8.                 <Grid.RowDefinitions>
9.                     <RowDefinition Height="*" />
10.                     <RowDefinition Height="*" />
11.                     <RowDefinition Height="*" />
12.                 </Grid.RowDefinitions>
13.                 <Grid.ColumnDefinitions>
14.                     <ColumnDefinition Width="100" />
15.                     <ColumnDefinition Width="*" />
16.                 </Grid.ColumnDefinitions>
17.                 <Image Source="
   {Binding weathercode, Converter=
   {StaticResource WeatherCodeToImageConverter}}"
18.                 Grid.RowSpan="3"
19.                 Margin="10" />
```

```

20.         <Label Grid.Column="1"
21.             Padding="5,0"
22.             Text="{Binding time}"
23.             FontAttributes="Bold"
24.             VerticalOptions="Center" />
25.         <Label Grid.Column="1"
26.             Grid.Row="1"
27.             Padding="5,0"
28.             Text="
29.             {Binding weathercode, Converter=
30.             {StaticResource WeatherCodeToStringConverter}}"
31.             VerticalOptions="Center" />
32.         <HorizontalStackLayout Grid.Column="1"
33.             Padding="5,0"
34.             HorizontalOptions
35.             =
36.             "StartAndExpand"
37.             Grid.Row="2"
38.             Spacing="20">
39.             <Label Text="
40.             {Binding temperature_2m_max,
41.             StringFormat='Max: {0:F1} °C'}"
42.             VerticalOptions="Center" />
43.             <Label Text="|" VerticalOptions="Cen
44.             ter" />
45.             <Label Text="
46.             {Binding temperature_2m_min,
47.             StringFormat='Min: {0:F1} °C'}"
48.             VerticalOptions="Center" />
49.         </HorizontalStackLayout>
50.     </Grid>
51. </DataTemplate>
52. </CollectionView.ItemTemplate>
53. </CollectionView>

```

Run the app now and check if the weather forecast for all seven days is shown in the scrollable list:

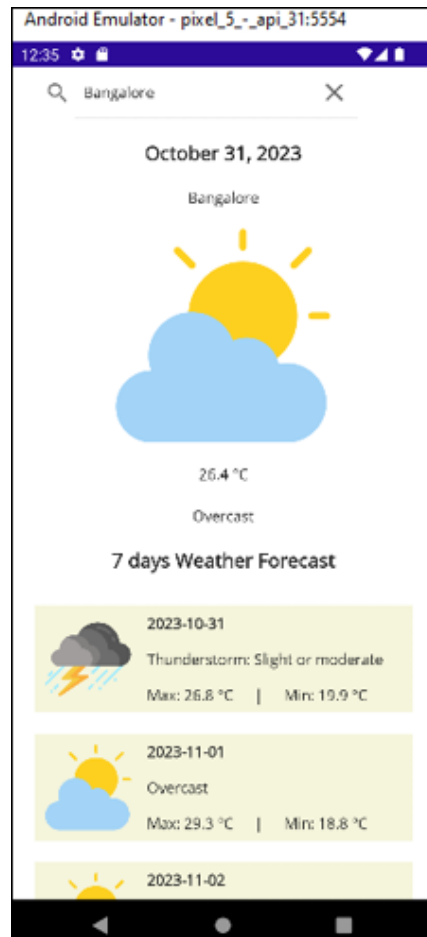


Figure 10.14: Screenshot of the app

Conclusion

In this chapter, you have developed your fifth and final project using .NET MAUI. You will now be able to run it on your device as well. If you face any errors, check the source code provided for reference, compare it with your code, and make the necessary corrections.

Points to remember

Here are some key takeaways from this chapter:

- Ensure that you choose the right .NET Framework version while creating the project.
- This app demonstrated how you can implement a multi-page app with a navigation system in .NET MAUI.
- The root page must be set in the App class.

- To navigate from the root page to the content pages, use the `PushAsync()` method in the Navigation class. Remember to use the `await` keyword and mark the wrapper function as an async method as well.
- The focus of this chapter was on .NET MAUI features and not on design patterns. For real-world implementation, you can explore how to use the MVVM pattern, which provides several advantages, including separation of concerns, testability, and easy maintainability.
- If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

- ahead-of-time (AOT) [17](#)
- Android Studio IDE [3](#)
- app development frameworks
 - cross-platform applications [4](#)
 - hybrid applications [3](#), [4](#)
 - low code/no code (LCNC) approach [4](#)
 - native apps [3](#)
 - web apps [3](#)
- application programming interface (API) [3](#), [15](#)
- Avalonia [10](#)
 - advantages [10](#), [11](#)
 - limitations [11](#)

B

- Body Mass Index (BMI) calculator app [131](#)
 - data binding , implementing [141](#), [142](#)
 - data binding to object [142-144](#)
 - data entry layout, creating [138-140](#)
 - layout, creating [132-137](#)
 - value converters, binding [144-148](#)

C

- C# [54](#)
 - arrays [74](#), [75](#)
 - class [67](#)
 - conditional statements [60](#)
 - constants [54-56](#)
 - delegates [72](#), [73](#)
 - events [73](#), [74](#)
 - expressions [59](#)
 - inheritance [69](#), [70](#)
 - interfaces [70](#), [71](#)
 - lists [75](#)
 - loops [62](#)
 - methods [68](#)
 - operators [59](#)
 - properties [68](#)

- string [56-58](#)
- types [54-56](#)
- variables [54-56](#)
- Color Picker app [93](#)
 - background color, changing [104](#)
 - color, copying to clipboard [108-113](#)
 - colors, specifying in XAML [101-104](#)
 - data entry section, creating [98-101](#)
 - layout, creating [94-97](#)
 - random colors, generating [105-107](#)
- common language runtime (CLR) [23](#), [84](#)
- conditional statements, C#
 - if-else statement [60](#)
 - switch-case statement [61](#)
- Contrast, Repetition, Alignment, and Proximity (C.R.A.P.) [124](#)
- cross-platform application development [1](#), [2](#)
 - advantages [4](#), [5](#)
 - app development approaches [3](#)
 - apps, versus websites [2](#), [3](#)
 - emerging technologies [12](#)
 - limitations [5](#)

E

- eXtensible Application Markup Language (XAML) [10](#), [77](#)
 - data binding [87-91](#)
 - markup extensions [84](#), [85](#)
 - namespaces [83](#), [84](#)
 - object names [84](#)
 - overview [78](#)
 - passing arguments [86](#)
 - syntax [78](#), [79](#)

F

- Flutter [7](#)
 - advantages [7](#)
 - limitations [7](#), [8](#)

I

- intermediate language (IL) [17](#)

J

- just-in-time (JIT) [17](#)

L

- loops, C#
 - break statement [65](#), [66](#)

- continue statement [66](#)
- do-while loop [64](#), [65](#)
- for each [63](#)
- for loop [62](#), [63](#)
- while loop [64](#)

M

- migration from Xamarin.Forms [24](#)
 - assisted migration, using .NET upgrade assistant [25](#), [26](#)
 - manual migration [25](#)
- Model-View-ViewModel (MVVM) [10](#)
- MVVM architectural pattern [170](#)
 - advantages [170](#), [171](#)

N

- NativeScript [8](#)
 - advantages [8](#)
 - limitations [8](#), [9](#)
- .NET Base Class Library (BCL) [16](#)
- .NET MAUI app structure [20](#)
 - layouts [21](#)
 - pages [20](#), [21](#)
 - views [21](#), [22](#)
- .NET Multi-platform App UI (.NET MAUI) [11](#), [12](#), [15-17](#)
 - evolution, from Xamarin.Forms [23](#), [24](#)
 - migration, from Xamarin.Forms [24](#)
 - platform integration [26](#)
 - working [18](#), [19](#)

O

- object-oriented programming (OOP) [67](#)
- operators, C#
 - assignment operators [59](#)
 - logical operators [59](#)
 - mathematical operators [59](#), [60](#)
 - relational operators [59](#)

P

- progressive web applications (PWAs) [3](#)

R

- React Native [6](#)
 - advantages [6](#)
 - limitations [6](#)
- responsive web design (RWD) principles [3](#)
- RGB Color Mixer [93](#)

S

Sky [7](#)

T

Tip Calculator app [115](#)

- calculation, performing [127-129](#)

- data entry section, creating [121-124](#)

- graphical design principles, applying in UI design [124-126](#)

- layout, creating [116-121](#)

U

unique device ID (UDID) [50](#)

Unit Converter app [151](#)

- conversions, performing [163-167](#)

- data entry section, creating [161](#), [162](#)

- main menu layout, creating [152-158](#)

- navigational system, creating [158-161](#)

Universal Windows Platform (UWP) [9](#), [24](#)

Uno Platform [9](#)

- advantages [9](#)

- limitations [10](#)

V

Visual Studio IDE [30](#)

- debugging, on Android devices [47-49](#)

- debugging, on Android Emulator [45-47](#)

- debugging, on Windows [40-44](#)

- environment setup, testing [33-40](#)

- installation [30-32](#)

- iOS device provisioning [50](#)

- Pair to Mac for iOS development [50-52](#)

W

weather app [169](#)

- custom type converters, creating [183-186](#)

- model, creating [179](#), [180](#)

- MVVM architecture [170](#)

- project structure, creating [171-174](#)

- seven-day weather forecast, displaying [188](#), [189](#)

- UI, creating [171-174](#)

- UI, designing [174-176](#)

- UI, updating place name [176](#), [177](#)

- UI, updating with date [176](#)

- UI, updating with weather data [187](#), [188](#)

- weather API, calling from code [181](#), [182](#)

- weather API, using [177](#), [178](#)

- weather forecast, updating [190](#), [191](#)
- Windows UI 3 (WinUI 3) library [16](#)
- WMO Weather interpretation code (WW) [183](#)

X

- XAML syntax [78](#), [79](#)
 - attached property syntax [82](#), [83](#)
 - attribute syntax [80](#)
 - comments [80](#)
 - empty element syntax [81](#)
 - object element syntax [80](#)
 - property value syntax [81](#), [82](#)

Table of Contents

Cover	1
Title Page	2
Copyright Page	4
Dedication Page	5
About the Author	6
About the Reviewer	7
Acknowledgement	8
Preface	9
Table of Contents	17
1. Introduction to Cross-platform Application Development	23
Introduction	23
Structure	24
Objectives	24
Introducing cross-platform app development	24
Apps versus. websites	25
App development approaches	25
Pros and cons of cross-platform app development	27
Popular development frameworks and technologies	29
React Native	29
Flutter	30
NativeScript	32
Uno Platform	33
Avalonia	34
.NET MAUI	36
Emerging technologies and cross-platform app development	37
Conclusion	37
Points to remember	38
2. Overview of .NET MAUI	40

Introduction	40
Structure	41
Objectives	41
Introduction to .NET MAUI	41
Supported platforms	44
Understanding how NET MAUI API works	44
.NET MAUI app anatomy	46
Pages, Layouts and Views	47
Pages	47
Layouts	47
Views	48
Evolution of NET MAUI from Xamarin.Forms	50
Migration from Xamarin.Forms	52
Manual migration	52
Assisted migration using .NET upgrade assistant	53
Platform integration	54
Conclusion	55
Points to remember	55
3. Development Environment Setup	57
Introduction	57
Structure	57
Objectives	58
Installation of Visual Studio IDE	58
Testing the environment setup	61
Debugging on Windows	71
Debugging on Android Emulator	76
Debugging on Android devices	78
iOS device provisioning	81
Pair to Mac for iOS development	81
Conclusion	84
Points to remember	84
4. A Crash Course in C#	85
Introduction	85
Structure	85

Objectives	86
Introduction to C#	86
Types, variables, and constants	87
Strings	89
Operators and expressions	93
Branches and loops	94
Loops	97
For Loop	97
Foreach	98
While and Do-while	99
Break and continue	101
Classes, methods, and properties	103
Classes	103
Methods	104
Properties	105
Inheritance and interface	106
Interfaces	107
Delegates and events	110
Events	111
Arrays and lists	113
Conclusion	114
Points to remember	115
5. Introduction to XAML	116
Introduction	116
Structure	116
Objectives	116
Overview of XAML	117
XAML syntax	117
Object element syntax	119
Comments	119
Attribute syntax	120
Empty element syntax	120
Property value syntax	121
Attached property syntax	122

Namespaces	123
Object names	124
Markup extensions	125
Passing arguments	126
Data binding	128
Conclusion	134
Points to remember	134
6. Project-1: Color Picker	136
Introduction	136
Structure	136
Objectives	136
Creating the layout	137
Creating the data entry section	141
Specifying colors in XAML	146
Changing the background color	150
Generating random colors	151
Copying the color to the clipboard	154
Conclusion	162
Points to remember	162
7. Project-2: Tip Calculator	164
Introduction	164
Structure	164
Objectives	165
Creating the layout	165
Creating the data entry section	172
Applying graphical design principles in UI design	175
Performing the calculation	179
Conclusion	183
Points to remember	183
8. Project-3: BMI Calculator	185
Introduction	185
Structure	186
Objectives	186
Creating the layout	186

Creating a data entry section	193
Data binding between UI controls	197
Data binding to an object	199
Binding value converters	202
Conclusion	208
Points to remember	209
9. Project-4: Unit Converter	210
Introduction	210
Structure	210
Objectives	211
Creating the main menu layout	211
Creating the navigational system	219
Creating the data entry section	223
Performing the conversions	226
Conclusion	233
Points to remember	233
10. Project-5: Weather App	235
Introduction	235
Structure	235
Objectives	236
Introduction to MVVM architecture	236
Creating the project structure and the UI	237
Designing the user interface	241
Updating the UI with date	242
Updating the UI with place name	243
Using the weather API	244
Creating the model	246
Calling the weather API from code	249
Creating custom type converters	251
Updating the UI with weather data	256
Displaying seven-day weather forecast	258
Updating the weather forecast	261
Conclusion	263
Points to remember	263

